CONDITION RECOGNITION FOR A PROGRAM SYNTHESIZER

Charles Wayne Miller

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

CONDITION RECOGNITION FOR A
PROGRAM SYNTHESIZER

by

Joseph Shawn Lape

and

Charles Wayne Miller

June 1981

Thesis Advisor: Douglas R. Smith

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Condition Recognition for a Program Synthesizer	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis: June 1981
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Joseph Shawn Lape Cahrles Wayne Miller	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School Monterey, California 93940	12. REPORT DATE: June 1981 13. NUMBER OF PAGES: 147
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

program synthesis, automatic programming, conditions, example computation, static processing, dynamic processing, miniterms, character set hierarchy, condition recognition

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

An enumeration algorithm which synthesizes programs from example computations is presented. The algorithm, originally proposed by Alan W. Biermann of Duke University, assigns a labelling of the instructions contained in an example trace consistent with producing minimum state Moore machine representations for the synthesized programs. Techniques for processing the information to reduce enumeration are given. Biermann's algorithm is extended by trace preprocessing techniques which identify and generalize

conditions on instruction sequencing in the synthesized programs without the user's assistance. The techniques are presented using text editing as the domain, but are general enough to be extendable into other domains.

CONDITION RECOGNITION FOR A PROGRAM SYNTHESIZER

by

Charles Wayne Miller
Captain, United States Marine Corps
B.E., Vanderbilt University, 1972

and

Joseph Shawn Lape
Captain, United States Marine Corps
B.S., University of Louisville, 1975

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1981

ABSTRACT

An enumeration algorithm which synthesizes programs from example computations is presented. The algorithm, originally proposed by Alan W. Biermann of Duke University, assigns a labelling of the instructions contained in an example trace consistent with producing minimum state Moore machine representations for the synthesized programs. Techniques for processing the information to reduce enumeration are given. Biermann's algorithm is extended by trace preprocessing techniques which identify and generalize conditions on instruction sequencing in the synthesized programs without the user's assistance. The techniques are presented using text editing as the domain, but are general enough to be extendable into other domains.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENT

We wish to acknowledge Alan W. Biermann for the extra help that he furnished us while we were doing the research for this thesis, and for the insights which he gave us on methods of programming by example.

I. INTRODUCTION

A. BACKGROUND

Since the introduction of electronic computing machines, manual tasks that are mundane, tedious and/or repetitious have been considered for automation. The computer is ideally suited for this type work since it neither complains of boredom nor wanders from its assigned task. The machine meticulously sequences through a series of computations over and over, producing answers consistent within the limitations of the hardware. As consistent as the computer is at performing tasks, assigning the tasks is still left to the user of the system.

Programming the early machines was a difficult chore. Communications between man and machine were only accomplishable through the language of the machine. This machine language consisted of binary coded machine operations. The efficient machine language programmer had to memorize these codes or keep a list of the codes close by. All control transfer points had to be coded in absolute machine addresses which the programmer calculated by hand. A programmmer had to interpret the binary representation of the machine operations to determine the cause of errors in programs. There were no diagnostic messages to aid the user in isolating errors. The difficulty of programming in

machine language led to a search to find better ways of generating programs. The first step was the recognition that the computer was a good bookkeeper, capable of computing absolute addresses from labels and translating mnemonic representations of machine operation codes. Webster's New Word Dictionary, Second Edition, defines mnemonic to be, "a system or technique of improving memory by the use of certain formulas." Soon programs were written which would accept abstract programs containing mnemonics and labels, convert the mnemonics into machine operation codes and translate the labels into absolute machine addresses. These programs produced executable machine language code as output. These translation programs were called assemblers and the data they translated were called assembly language programs.

Assembly language provided some automation of the manual tasks associated with machine language programming. An important convenience of assembly language is the readability of the programs when compared to machine language programs. The mnemonics convey the meaning of their function while the labels relieved the programmer of calculating absolute addresses for control transfer points. Assembly language provided a level of abstraction which allowed programmers to concentrate on the programming problem without dealing with every atomic machine operation. The assembler provided bookkeeping, address translation and

mneumonic decoding fast and efficiently. Programmers were now capable of producing more code in less time with fewer errors with assembly language.

Assembly language eased the programmers task but it still could not be considered a panacea for computer-human interaction. Assembly language still required the programmer to maintain control over many machine operations and he had to provide the logic to control the flow of program execution. The instructions used to perform control functions appears as similar code fragments in most programs written in assembly language. These code fragments performed fuctions such as controlling branching decisions and keeping count of loop indices. When it was observed that common code fragments appeared across a wide range of assembly programs, it was recognized that these code fragments could be represented as a single instruction and the computer could translate the single instruction into the code fragment it represented. The programs that translate these complex instructions are called compilers or interpeters. The complied or interpeted languages that followed assembly language in this evolutionary process incorporated the program fragments as a single instruction for the language. Constructs such as FOR, DO WHILE and IF THEN are examples of higher level control structure implementation.

FORTRAN was the first in a long line of higher level languages. FORTRAN differed from the others by becoming

endeared to a family of users and the language endures today as one of the most frequently used higher level languages. What qualities of the language produced this popularity?

The FORTRAN language is attributed to John Backus. His primary goal when designing the language was to make the language resemble the notation used in high school algebra. Since the notation used in high school algebra was familiar to a wide audience, FORTRAN gave a friendly appearance. The language's apparent simplicity is the endearing quality of FORTRAN. Some other language implementors failed to recognize this point and their languages never received wide acceptance. ALGOL is an example of a powerful language that never received the acceptance anticipated.

Other programming languages that followed added compact representation of other recurring program fragments. The higher level constructs were not limited to control structures but also included constructs for data manipulation functions. Iverson's [1] APL (A Programming Language) provided powerful operators capable of performing complex functions such as matrix multiplication in one instruction.

This trend continues today. Many of the newer languages implement sophisticated and powerful operators and control structures. Some of these languages are for a select segment of computer users, intended for application to a particular domain. The users are expected to be familiar with the

domain, so the form of the language should be familiar to the user also. A problem with a domain specific language is its inability to adapt to other areas. To work in another area the user must become familiar with another language. A phenomenon demonstrated by many computer users is a reluctance to adapt themselves and learn a new language that may be more appropriate for a given task. Either they break the egg with a sledge hammer or dig the well with a spoon. When required to use a new language, the user will likely use only a small subset of the language that is capable of doing the job. Worst than using only a subset of the language features is the tendency to bring old programming styles applicable to the old language into the new language. The point that is to be made is that learning a new programming language is a hard chore and is avoided whenever possible.

Another direction which the automation of programming tasks has taken is the development of a programming environment. A programming environment automates some of the manual chores by providing the user with aids that assist him in constructing programs. The environment includes a programming language, an interactive syntax-directed editor and an on-line debugger. The editor provides syntax error diagnostics while the programmer is creating the source file. The programmer is forced to correct the syntax error immediately before the editor will allow him to continue

programming. The error should be readily apparent to the programmer because it is in the latest input. The on-line debugger allows the programmer to actively test his program, halt execution, check the value of variables, change the value of variables or change the code itself. Program environment systems may even allow the programmer to switch from the the editor to the on-line debugger and back at any time. A programming environment can be summarized as a friendly interface utilizing an intelligent editor which can recognize syntax errors in the associated programming language and one that contains other interactive programming tools.

Programming has been called an art form requiring intellectual creativity. The automation of intellectual behavior is a field of study within Computer Science called Artificial Intelligence. The study of the automation of programming tasks which require human-like reasoning is called Program Syntnesis or Automatic Programming. It is not our intention to provide a definition of intelligent behavior for a machine since there is considerable disagreement even among the experts. However, we note that the goal of research in automatic programming is the same goal that led to all the advances in programming languages. Informally, this goal is to make the interaction between man and computer as painless as possible. That is, painless for the man but not necessarily for the computer. Dijkstra [2]

objects to our automation of programming by claiming, "We should not automate programming even if we can, because it would take away our enjoyment of the task." We note there are those who may require the use of computer services that have neither the time nor inclination to obtain the required education to do that chore. These include professions such as lawyers, physicians, and even theoretical physicists. We assume, if programming becomes fully automated, the programmers will then turn their attention toward other creative and stimulating pursuits. R. Hamming has said, "The purpose of computing is insight not numbers."

Many on-going efforts are aimed at providing better systems for the user so he may create programs faster, with less errors and with less effort. The history of programming language development has shown that automation of many programming tasks is feasible. How much more of the programming tasks can be automated? What would be considered the ultimate system for producing computer programs?

B. AUTOMATIC PROGRAMMING

1. General

Program synthesis or automatic programming is a research topic concerned with the development of systems that provide more and more automation of the programming process, particularly those tasks requiring human-like reasoning. The goal is not to create systems that program themselves, but to create systems which can construct, under

the direction of a user, programs that can perform some function he desires. These systems must be easy to use, easy to learn, and increase the efficiency of the user. The users of these systems will no longer be restricted to the few computer professionals, but will include other professional fields as well as non-professionals. Automatic programming systems are to interact with the user, recognize requirements, and then synthesize a correct program that satisfies the requirements.

Two questions arise in the research on automatic programming. First, what is the form of the interaction between the user and the system? This question is called the specification problem because it is concerned with issues relating to how the user is to inform the system of his requirements. The second question is, given a specification method, what synthesis technique is available to be applied that will transform the specification into an appropriate program. The technique used for synthesis is often dependent upon the form of the problem specification and most of the projects involving automatic programming consider both problems together. It has been proposed by Green [3] that the two questions should be separated with research proceeding concurrently on both problems. He proposes there is a standard intermediate representation of the problem specification which would permit interaction between the two problems.

Four techniques have been proposed for the specification problem which dominate the literature on automatic programming. Each of the proposed techniques of problem specification introduce a different approach to the synthesis problem. The four specification techniques can be categorized as follows:

1. Natural Language.
2. Formal Problem Specification.
3. Input-output Pairs.
4. Example Computations.

Each of these specification techniques will be dicussed in the following subsections and the relationship to a synthesis approach will be discussed.

2. Problem Specification with Natural Language

A visionary approach to the specification problem is the use of natural language. Natural language provides a fast, comfortable method of communication which is already understood by humans. Implementation of a natural language understanding system has proven to be a very difficult problem (Glass [4]).

Two forms of natural language are the spoken form and the written form. Understanding spoken language increases the degree of difficulty because the communication is in the form of audio waves. Once the audio input is captured, it must be converted into another form for further syntactic and semantic analysis. The reader will note that

once the audio input has been captured and converted the problem of written and spoken language becomes the same. That is, the internal representation of the spoken and written word can be the same and the problem becomes one of inferring meaning from the representation. Future advances in voice understanding hardware can be expected and these advances may be expected to find their way into use.

A complete natural language understanding system would be expected to be able to understand all grammatically correct sentences. However, natural languages do not have finite grammars. This complexity implies a complete understanding system cannot be implemented. However, a system capable of understanding a subset of natural language can prove useful in specific domains. Early examples of programming through natural language dialogue is presented in a survey by Heidorn [5]. Current work on understanding natural language may be found in Biermann [6], and Walker [7].

In conclusion natural language understanding is a difficult problem that can be solved only in limited domains. The use of natural language in programming has been shown to be possible by Heidorn [5], and by Biermann [6] in limited domains. The systems developed up to today have been experimental systems and the results will aid in understanding the problem. Natural language programming systems will not be available for industry for at least a

decade. Finally, we present the example Biermann [6] describes as a natural language specification for a problem. This example is quoted from his paper on natural language programming. Its intent is to give a feel for programming in natural language. This example does not specify the algorithm that is to be used although a natural language programming system would be capable of accepting such a specification.

> "When I ask for a status report on a doctorial student, give me his or her year in grad school, source and amount of financial support, and which core exams have been passed. If the student has begun a thesis give me the advisor and thesis topic."

3. Formal Problem Specification

The second technique is formal specification of the problem. As the name implies, the input is in a more rigid structure than natural language. This technique allows the user to convey the behavior he desires the synthesized program to have without specifying the algorithm that is to be used. Smith [8] gives the following definition for the form of a formal specification of a problem A.

> "A(x) = z such that z c S & P(z,x) where x c D & I(x) where D and S are the input and output data types respectively, and I and P are the input and output conditions respectively."

An example of a formal problem specification for a program to compute the integer square root of a nonnegative integer n may be found in Manna and Waldinger [9].

```
"sqrt(n) <== FIND z SUCH THAT
        integer(z) & z**2 =< n < (z + 1) ** 2
        WHERE integer(n) & 0 =< n"
```

In the above example n is an element of the input data type, z is an element of the output data type, sqrt is the problem name, integer(n) & 0 =< n is the input condition, and integer(z) & z**2 =< n < (z+1) ** 2 is the output condition.

Formal problem specification and its application to the program syntnesis problem can best be explained through examination of the work by Manna and Waldinger [9], Manna and Waldinger [10], and Smith [8]. Although all of the work is similar in that the formal specification is changed into an appropriate program by some form of rewrite. It is valuable to differentiate the approaches by their rewriting methods.

The first example is the system of Manna and Waldinger [9]. Their system, called a deductive approach, converts the formal specification into a program in some target language. Their approach, "combines techniques of unification, mathematical induction, and transformation rules into a single system." The following is an brief explanation of this conversion.

A structure is needed to contain initial and intermediate results of the conversion process. This structure is call a sequent. The sequent is a tableau containing two lists. The first list is a list of assertions and the second list is a list of goals. Each element in

either list may have an output expression associated with it. Figure 1 represents a sequent as a table. Each row in the table may contain either an assertion or a goal but not both. Figure 1 is the initial sequent for the integer square root problem given above. The input condition has been placed in the assertion list and the output condition placed in the goal list. The output variable is associated with the output condition in the output expresssion column. This initiation action assumes the input condition is true and a search is attempted for the truth of the goal or output condition.

```
sqrt(n) <== FIND z SUCH THAT
            integer(z) and z**2 =< n
            and n < (z+1) ** 2
            WHERE integer(n) and 0 =< n
```

Assertions	Goals	Output sqrt(n)
integer(n) and 0 =< n		
	integer(z) and z**2 =< n and n < (z+1)	z

Figure 1. Initialized Sequent for the Square Root Problem

During this search if the sequent ever contains a row where the assertion can be trivially shown to be false or the goal shown to be true and if the output expression for that row contains only primitives from the target language then the output expression is taken as the desired synthesized progam.

Once the tableau is initialized, the system's deductive rules are applied to the assertions and goals. The application of these rules will cause the creation of new assertions and goals and associated output expressions. The rules may then be applied to the new goals and assertions until the condition for a program is satisfied. The application of the rules change th entries in the tableau without changing the meaning of the tableau. We recommend that the interested reader review the original work for a description of the rules and their application.

The attraction of this theorem-proving technique is that the resulting program can be proven correct by the same steps used to create it. Currently there is not a running implementation of this technique. One of the implementation questions is determining what rule to apply at each step in the synthesis process. This problem can be viewed as a search through all possible sequences of rule applications. This search space may become astronomical for any relatively complex program since it may require hundreds of rule applications. What is needed is a mechanism that can control

the search in a reasonable fashion. The form of control may be heuristic in that there is a feel for where a rule should be applied. If this intuitive feel can be quantized, then this technique may become practical.

Earlier work by Manna and Waldinger [12] on the DEDALUS automatic programing system also required formal problem specifications. The DEDALUS system, an implemented automatic programming system, utilized only transformation rules. A tranformation rule simply rewrites a portion of the specification into another equivalent form. The continuous application of these rules would eventually result in a program in the target language.

4. Input-Output Pair Specification

Input-output pairs is a method of describing a problem with examples of input and output behavior. For example, if someone wanted to describe a program to compute the Fibonacci numbers then he could supply the input-output pairs.

(1, 1)
(2, 3)
(3, 5)
(5, 8)
(8,13)

The goal of a synthesizer system is to determine the desired program from the examples of the input-output behavior. One approach is to enumerate all possible programs in the target language in order and test each program for the desired behavior. That is, test each enumerated program

by giving it the input from each of the examples and see if the program will give the associated output. The enumeration will produce the correct program at some point but you cannot determine if an arbitrary program can produce the desired behavior (see Biermann [11]). Therefore, the following theorem is given by Biermann, "The programs for the partial recursive functions cannot be generated from sample of input-output behavior." A large class of programs may be inferred from examples of input-output pairs provided they belong to the class of programs where the halting problem is decidable. Smith [12] and Summers [13] have looked at the synthesis of LISP programs for example input-output pairs. It has been shown that a restricted class of LISP programs can be synthesized from example pairs without enumeration over the class. The reader is invited to review Biermann [14] and Gold [15] for theoretical background information.

5. Example Computations

Program specification using example computations allows more information to be obtained from the user. An example computation is a sequence of instructions, without an explicit control structure, which the user provides the system in order to describe the behavior he wants from a program. Examples are a good communication method which people use to describe new concepts or explain new processes. To describe a problem to the computer the user

uses the available instructions and provides an example of what he wants done. Figure 2 shows an example computation that demonstrates how to compute the first 10 Fibonacci numbers.

In Figure 2 the two operand instructions (MOV, ADD) perform the action on the two operands and leave the result in the first operand. For example, if A = 2 and B = 3 then ADD A,B would result in A = 5 and B = 3. All of the instructions perform action on some variables except for the START, HALT, and NOTE instruction. START and HALT flag the begin and end of the program respectively. The NOTE instruction is providing information on the reason for the execution of the next instruction.

This method of specification depends on the user to supply more information about the problem, including the algorithm to be synthesized. The algorithm is implicitly defined by the example computation that is given. This specification technique should be contrasted with the previous techniques. Note that the formal specification and the input-output pair specification only required the user to specify the desired behavior without specifying the algorithm. Thus it can be claimed that these two methods intentionally ignore information that the user has, assuming that most users have an idea of the form of the algorithm.

```
START
MOV    A,1
MOV    B,1
MOV    C,10
PRINT  B
DCR    C
ADD    A,B
PRINT  A
DCR    C
ADD    B,A
PRINT  B
DCR    C
  .
  .
  .
PRINT  A
DCR    C
NOTE   C =< 0
HALT
```

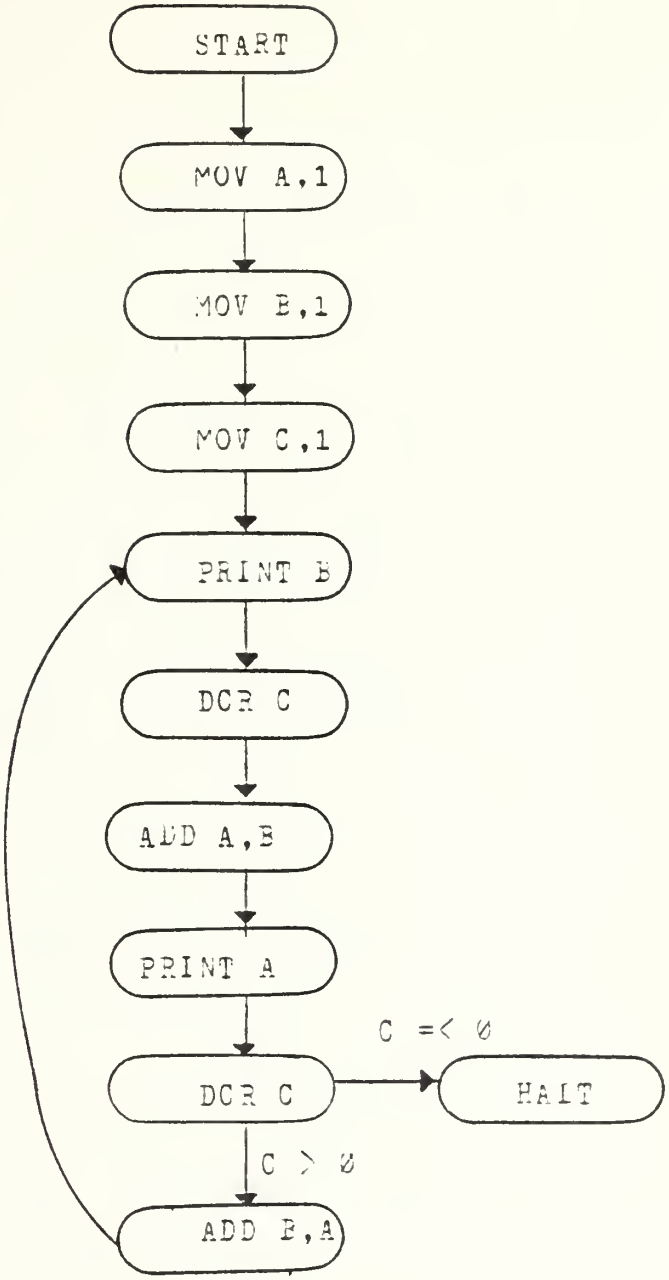


Figure 2. An Example Computation

The primary contributor to the understanding of program synthesis has been Alan W. Biermann (see Biermann and Krishnaswamy [16] and Biermann, Baum and Petry [17]). In particular, Biermann [16] provides a formal definition of an algorithm that will synthesize programs from example computations. The algorithm and variations have provided the basic structure upon which this thesis has been developed. Briefly, the algorithm identifies the conditions that may have inadvertently (or purposely) been left out of the computation. A condition is a predicate as defined in predicate calculus. That is, an entity for which a truth value may be measured. Once the omitted conditions have been inserted, the algorithm finds a labelling for the instructions such that a program with a minimum number of instructions is produced. To explain this labelling, assume the instruction ADD A,B appears in three different locations in an example computation (see Figure 2). Suppose it was known that there has to be two occurrences of the instruction. Then two of the instructions could be labeled with a 1 and the other instruction labeled with a 2 to indicate that the instruction labeled 2 is different from the instructions labeled 1. Finding the labels for the instructions in the example computations requires an enumeration search of all possible labellings. The labelling selected is the first labelling that produces a program that is deterministic.

This algorithm is complete and the synthesized programs are sound. Completeness means that the algorithm can synthesize every possible program. Soundness mean that the synthesize program will correctly execute the example used to construct it. A disadvantage of this synthesis method is the algorithm is an enumeration search and in the worst case will require exponential time on the length of the example computation to find a solution. Techniques have been developed to speed up this search that will produce satisfactory response for most pratical programs.

6. A General Automatic Programmer Design

Before leaving this section on automatic program we wish to discuss a design for an automatic programmer that uses at least two of the specification techniques. The name of the system is PSI and was designed by a group of researchers at Stanford's Artificial Intelligence Laboratory. The research effort was headed by Cordell Green [3]. Green has presented a high level design of an autoprogrammer that identifies some of the more important areas that need further research. Green admits that the design was an effort to focus attention on some of the sub-areas of the overall synthesis problem. His modular design does focus attention on different aspects of the problem. The design decision to split the overall problem into two main sub-problems of acquistion and synthesis is of particular interest. This design choice allows work to

proceed concurrently on two hard problems with the interface between the problems being some intermediate representation of the problem.

PSI is a knowledge-based program understanding system organized as a collection of interacting modules. Figure 3 details the high level modular design of the PSI system. The PSI design divides the system into two groups. The acquisition group interfaces with the user and collects the specification given by the user while the synthesis group produces a program in some target language that meets the user's requirements. Communications between the two major groups is through an intermediate representation called the program model. The goal of the acquisition group is to accept the user's specification by either natural language dialogue or by traces, and present a unified entity to the synthesizer group. The implementation of the synthesizer group is then simplified because of the consistent representation it receives. Since the user's input is converted into an intermediate representation that is supplied to the synthesizer group, the user is free to switch from one specification technique to another during program specification.

The overall interaction with the user is meant to be through natural language dialogue. Since natural language understanding is not currently within the

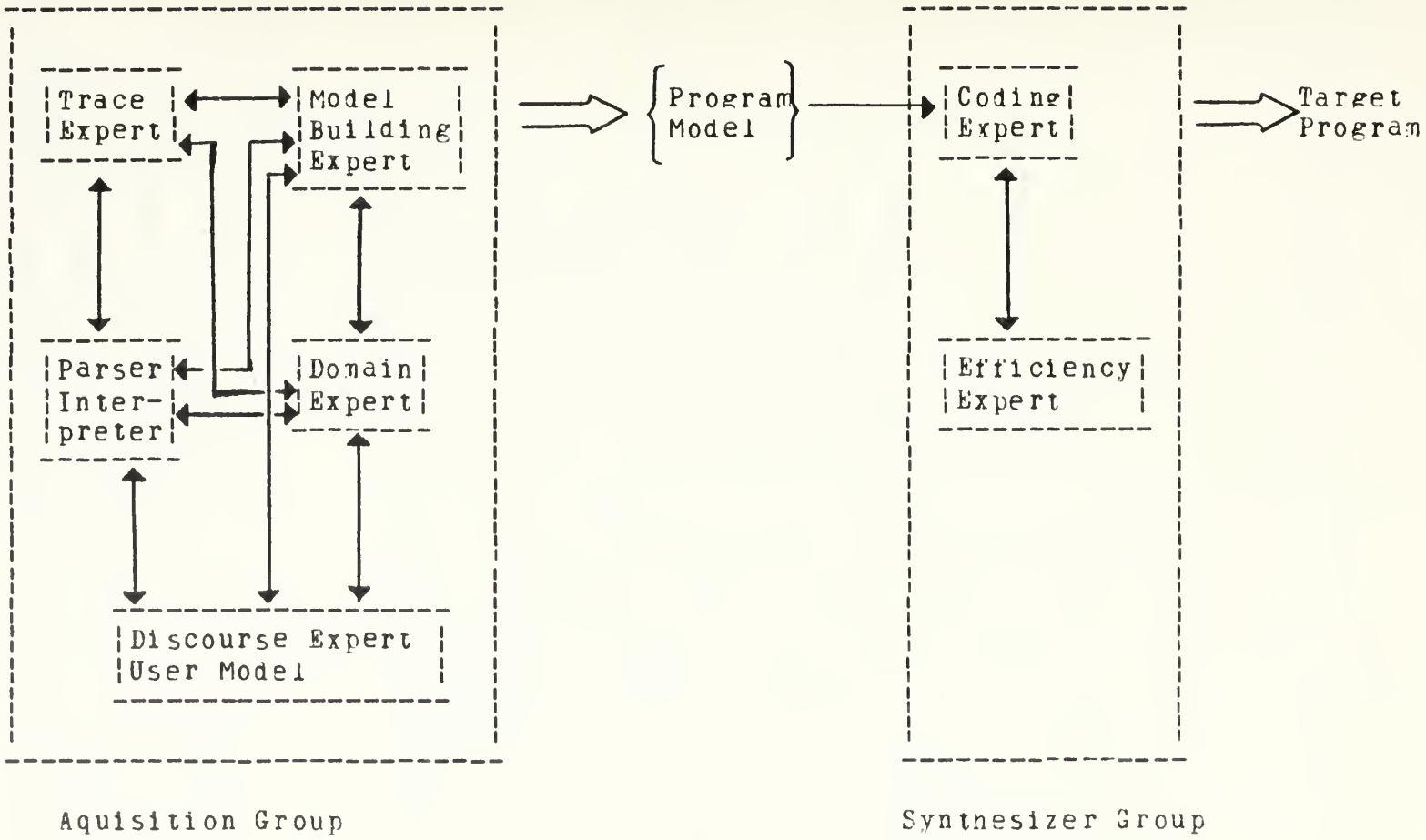


Figure 3. PSI's Modular Design [3,p.6]

state of the art, the system must interact in a subset of natural language limited to a particular domain.

The system-user interaction is to appear as natural as possible. The system has been designed to include a mixed-initiative dialogue capability which means the user or the computer can assume the dominant communication role at different times during the discourse. This allows the user to provide as much knowledge as he can to help the synthesis process and allows the computer to assist the user by asking questions or providing responses. The system develops a current model of the user and a model of the context that assists the system in determining when to assume the initiative and what questions to ask the user.

A partial implementation was completed in 1976 that included the synthesis expert and the efficiency expert from the synthesis group. The acquisition group modules have proven to be a more difficult assignment and only portions of the acquistion group have been implemented. The important point of the PSI design is that it provides a modular division of the program synthesis problem that helps provoke study into these sub-problems.

C. OBJECTIVES

Automatic programmers, which synthesize programs from example computations, require conditions to be explicitly defined by the user in order to generate programs with a minimum number of instructions. Previous work (Biermann and

Krishnaswamy [16], and Biermann [18]) has reduced the number of required conditions, but has not eliminated the need for the user to explicitly state a minimal set of conditions.

The explicit definition of conditions is not a natural part of an example computation. That is, one would not normally give control structure information when using examples to explain how a task is to be performed. Our objective is to provide an environment where the user may define the tasks he wants accomplished without explicitly defining the control structures that specify the flow of execution in a synthesized program.

We will implement an automatic programming system based upon the example computation specification method in order to study the feasibility of identifying conditions from user actions. We limit this study to the domain of text editing in order to provide a well defined area in which to work. It is hoped that the results of our efforts may provide insight into the overall problem and generate further research which will extend condition identification to other domains.

D. THESIS ORGANIZATION

The thrust of this thesis is the developement of methods for the automatic construction of conditions necessary for the proper synthesis of programs from example computations. Example computation is one approach to the problem of program synthesis. Chapter One introduces the reader to

program synthesis and gives a brief historical perspective of the evolution of this field of study. Chapter One also provides a comparison of the different proposed approaches to this problem.

An automatic programmer has been implemented to support this research. This synthesizer was developed to use the example computation method for program specification. Chapter Two is a detailed explanation of our particular implementation. Chapter Two includes a discussion of techniques we have incorporated in our implementation which speed up the synthesis process.

Chapter Three presents our approach to generating conditions given an example computation. It describes algorithms which will generate conditions from a sequence of editor instructions.

Chapter Four discusses the result of our research. A brief discussion is included on the merits of the synthesizer which we have implemented and recommendations are given for potential improvement. Finally, Chapter Four presents a review of our work on identification and construction of condtions from example computations. Areas requiring further research have been highlighted and examples of possible applications to other domains have been pointed out.

II. SYNTHESIZER

A. GOALS

There is a two-fold purpose behind designing and building the program synthesizer. The first directly relates to the usefulness of the synthesizer. It is hoped that by "laying the groundwork" for an autoprogramming system, the impetus will be provided that will eventually result in a total automatic programming environment being available for the user. This environment is envisioned as an interactive one consisting of several components: an interface to provide the user with the means to perform example computations, a link between the interface and the synthesizer which records the user actions and transmits a trace of those actions to the synthesizer, the synthesizer itself which produces the algorithm in some internal form, and, finally, a translator that receives the internal representation of the algorithm and translates it into machine-readable form and/or user-readable form. The second purpose for which the synthesizer is built is to provide a suitable vehicle to be used in the main area of research that this thesis explores. If an autoprogrammer can generate correct algorithms from example computations, how much can be done to relieve the user from having to include branching or looping conditions in his example computations?

B. OVERVIEW

1. General Description

An automatic programming system which produces programs based upon the user's input of example computations has a natural appeal. Example computations are sequences of instructions performed in an algorithmic manner. For instance, if the user is doing a matrix multiply, computing the entry for the resultant matrix involves the sum of products from the appropriate row and column of the multiplicand and multiplier matrices, respectively. When humans communicate ideas to each other, the proper use of example computations often plays a vital role. It is hard to imagine trying to explain the method of multiplying two matrices together, or trying to explain the concept of set-subset relationships without being able to draw examples that enhance the explanations. This method of communication seems to be vital to human understanding of algorithms. Since programmers often use small example computations while coding programs, it seems that a logical approach to automatic programming would consist of the machine doing the actual program synthesis based upon example computations given by the programmer.

Program synthesis is the act of putting instructions together in such a way that an algorithm is built which accomplishes a desired task. Obviously, an algorithm which is an exact replication of the sequence of instructions will

accomplish the task, but it is uninteresting since it cannot be generalized to accomplish a set of related tasks. For example, a linear sequence of instructions which multiplies two 2 x 2 matrices together will only work for 2 x 2 matrices; however, by allowing loop constructs and if-then constructs, an algorithm can be produced which performs the more general task of multiplying any two matrices with legal row and column dimensions. So, in the case of the matrix multiply, the task of the program synthesizer is to produce a general matrix multiply algorithm given the example computation for a 2 x 2 matrix multiplication in some form such as:

```
c[1,1] = a[1,1] * b[1,1] + a[1,2] * b[2,1]
c[1,2] = a[1,1] * b[1,2] + a[1,2] * b[2,2]
c[2,1] = a[2,1] * b[1,1] + a[2,2] * b[2,1]
c[2,2] = a[2,1] * b[1,2] + a[2,2] * b[2,2]
```

Generalizing from the example computation also requires some means of noting when the array bounds have been reached for this example. In other words, conditions have to be interposed between some instructions where a change in the flow of control for the algorithm is necessary. An <u>input trace</u> is defined as a sequence of instructions and conditions which describes the example computation. In the matrix multiply example this might be accomplished thusly:

```
                C[1,1] = ∅
C[1,1] = C[1,1] + A[1,1] * B[1,1]
C[1,1] = C[1,1] + A[1,2] * B[2,1]

COND - col index of A = col size of A

                C[1,2] = ∅
C[1,2] = C[1,2] + A[1,1] * B[1,2]
C[1,2] = C[1,2] + A[1,2] * B[2,2]

COND - col index of A = col size of A

                  .
                  .
                  .
C[2,2] = C[2,2] + A[2,2] * B[2,2]

COND - row & col index of C = Dimension of C

                STOP
```

The program synthesizer used for this thesis is designed around concepts and ideas on syntnesizing a program given example traces as described in reference [17]. Previous researcn, references [16], [17], and [18], seems to indicate that correct programs can be synthesized on the basis of relatively few sample computations, but tnat tne amount of time required to do tne synthesis grows very quickly as a function of program complexity.

2. Trace Coding

Tne syntnesis procedure is domain independent; tnat is, the input trace can be coded into any consistent representation, and it will not affect tne operation of the synthesizer. Since tne synthesis procedure is independent of the input trace representation, alphanumeric characters will be used to represent instructions and conditions. They are distinguished from each other by their position within the

trace rather than by their symbolic representation. For example, an 'a' might represent an instruction or a condition. Within the instruction set itself, identical instructions are encoded as identical symbols. A simple trace of a routine to find all positive numbers in an input stream might be:

```
          A = Ø
          READ B

COND - B is negative

          A = A + 1
          READ B

COND - B is negative

          A = A + 1
          READ B

COND - B is positive

          PRINT B
            .
            .
            .
```

If the instruction A=A+1 is represented by a 'b', each occurrence of that instruction in the trace will have to be represented by a 'b'. The reason for this constraint is obvious. Since the synthesizer only receives a trace of the example execution, it cannot determine whether A=A+1 is the same instruction being encountered repeatedly in a loop, as it is in this example, or whether there are several independent occurrences of A=A+1. Figure 4 is an example of a typical coded input trace. The left-hand column entries are conditions and the right-hand column entries are

instructions. Figure 4 is read as state 's' transistions on condition 'x' to state 'a' which in turn transitions on 'x' to state 'b', and so forth.

transitions	states
-	s
x	a
x	b
x	c
x	b
y	e
x	c
x	b
x	c
y	s
y	a
x	b
y	d
x	b
x	f
x	d
x	b
x	f
x	d
y	s

Figure 4. Input Trace

3. Input/Output Trace Representation

A Moore-type representation, as defined in [17], can be used to highlight certain features that must be dealt with when producing an algorithm from an example trace. Throughout the rest of the discussion, Moore machines and algorithms will be used synonymously. Conditions relate to transitions and instructions relate to states of the machine. In fact, the function of the synthesizer can be viewed as that of determining a minimum-state deterministic

Moore machine equivalent of a non-deterministic Moore machine. Representing input traces as Moore machines will often show the non-deterministic structure of the example trace. This non-determinism must be resolved by the synthesizer in order for an algorithm to be generated. Figure 5 is the Moore machine representation of the input trace of Figure 4. Notice that at node 'b', the trace is non-deterministic. Transition 'y' leads from node 'b' to two different nodes; similarly, transition 'x' leads from node 'b' to two separate nodes. Figure 6 is the deterministic Moore machine which has been constructed by our synthesizer based upon the input trace given in Figure 4. The non-determinism has been resolved by splitting state 'a' into two states distinguished from each other by an integer prefix label. The assignment of the prefix label is the mechanism used by the synthesizer to prevent non-determinism. In order to accomplish this assignment, the synthesizer uses an enumeration technique. Each instruction is assigned a prefix label in a manner that maintains determinism and assures that the algorithm will correctly execute the input trace. It is easy to verify that the deterministic Moore machine of Figure 6 will execute the trace.

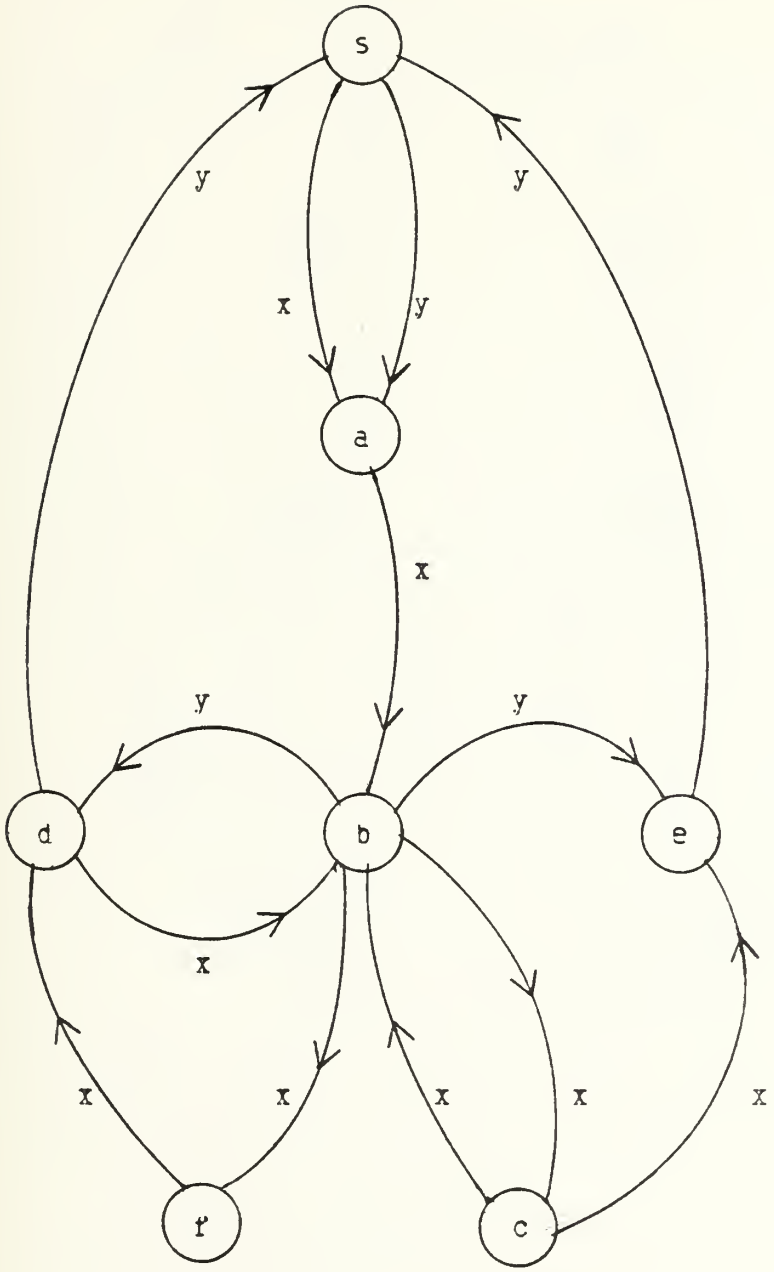


Figure 5. Non-deterministic Moore Machine

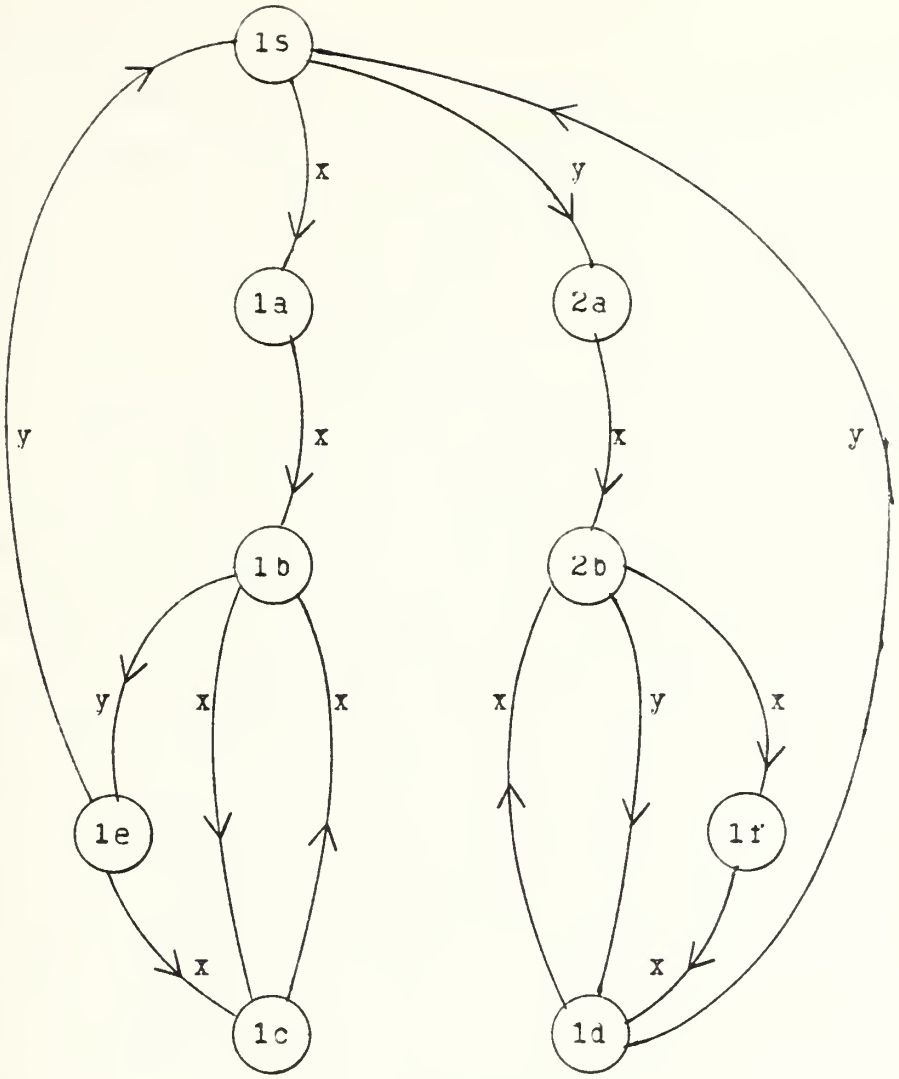


Figure 6. Deterministic Moore Machine

C. SYNTHESIS PROCEDURE

1. Function

The function of the synthesizer program is to provide a minimum-state, correct program consistent with the input trace of the example computation. The synthesis process will be completed when it is determined which occurrence of a labelled instruction corresponds to each particular instruction in the input trace. In order to accomplish this goal, the synthesizer is basically structured as a depth-first search algorithm. Backup and fixup mechanisms exist to enhance the search procedure when pruning has not kept the algorithm from traversing a fruitless branch of the search tree. The search mechanism attempts to assign a label to each instruction in such a manner that the generated algorithm remains technically correct; that is, nondeterminism is not allowed to exist and the original trace can still be executed. A number of techniques exist within the synthesizer which aid pruning of the search tree, and thereby make it possible to synthesize more complicated programs in a reasonable amount of time than could otherwise be expected from a general enumeration technique. These techniques offset the major disadvantage of exponential growth of the search space as a function of input which is found in a general enumerative search technique.

2. Concepts

Certain definitions and concepts must be presented before the actual algorithm is discussed. In order to facilitate the discussion, it is necessary to refer to Figure 7. Each level in the figure consists of an instruction-condition-instruction triple, referred to as an I-C-I. In Figure 7 the leftmost symbol under I-C-I is referred to as the leading instruction of the triple, the middle symbol is the condition, and the rightmost symbol is the trailing instruction. The trailing instruction at level i becomes the leading instruction at level i+1. So this input trace represents the instruction-condition sequence 's r a n s r a ...'.

level	I-C-I
1	sra
2	ans
3	sra
4	ada
5	axa
6	aya
7	axa
8	anr

Figure 7. Instruction-Condition-Instruction Triple

Two levels i and j are said to belong to the same couple-class if the elements of the level are the same.

Instruction elements of the trace which are in the same couple-class may be assigned the same prefix label during synthesis if the assignment does not cause non-determinism. For example, given the trace in Figure 7, levels 1 and 3 are in the same couple-class, as are levels 5 and 7. Difference set relations are another situation that can exist which is of interest. The first two elements of level i and level j are the same, but the third element is not the same. A difference set relation indicates that the leading instructions cannot be represented by the same state regardless of the prefix label assigned during synthesis because the leading instruction has the same transition to two different trailing instructions. Again using the above trace, level 2 and level 8 fall into this category. In this situation, the index 8 would be entered into the difference set for level 2. By implication, the index 2 is also in the difference set for level 8, although, in practice, it is not entered.

Once the initial couple-class information and difference set information have been determined, additional difference set information can be obtained through the chaining nature of differencing. For example, suppose the trace consists of the one shown in Figure 8. Then the Moore machine representation of this trace is shown in Figure 9.

index	trace
.	.
.	.
.	.
5	axa
6	axa
7	ays
.	.
.	.
.	.
8	axa
9	axa
10	ayt

Figure 8. Chaining of Difference Set Relations

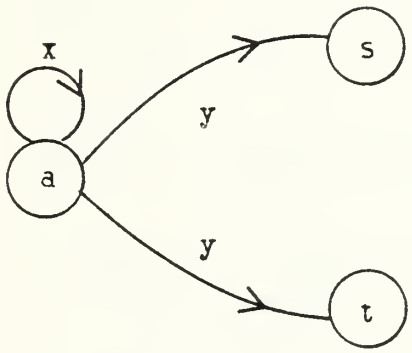


Figure 9. Non-deterministic Input Trace

This machine is obviously nondeterministic since state 'a' transitions by 'y' to two different states. Difference set resolution requires that the index for 'ayt' be in the difference set of 'ays'. Since that requirement causes different states to represent the 'a' in 'ayt' and in 'ays', and further since the trailing 'a' in the preceding level is exactly the same instruction, the preceding levels now satisfy the difference set relation. The leading

instruction and the condition are the same, but the trailing instruction in the I-C-I triple is different since they have previously been assigned to a difference set relation. Therefore, the lead instruction must be labelled with a different prefix during assignment and similarly, the levels above them. So the Moore machine will now be deterministic and in the following form.

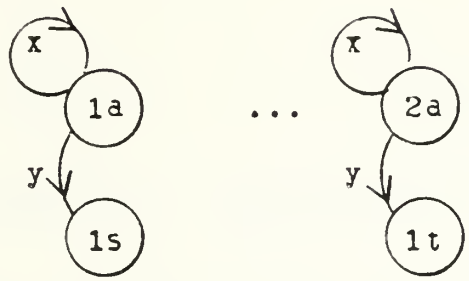


Figure 10. Deterministic Trace

Given a partial trace derived from the example execution, there are numerous Moore machines that can be constructed to satisfy the trace. At one end of the spectrum, a program can be constructed such that each succeeding state is assigned a different prefix label. This method always results in a straight-line program. Each instruction has one transition entering it and one transition exiting from it. Allowing this method produces the maximum size program consistent with the input trace. See Figure 11. This is not a particularly desirable method since it does not recognize loop structures that can significantly reduce the size of the program. Additionally,

it hides the basic structure of the algorithm. The major advantage, of course, is that absolutely no search is required to produce a deterministic machine.

condition	instruction
-	a
x	a
x	a
x	a

Figure 11a. Trace

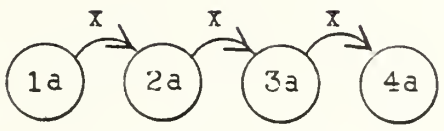


Figure 11b. Program

Figure 11. Straight-line program

On the other end of the spectrum, a program can be constructed such that each identical instruction receives the same prefix label. This method takes full advantage of loop structures, and will result in a minimum state machine. However, such a method will seldom produce a deterministic machine; therefore, it will not produce a satisfactory algorithm. See Figure 12.

level	cond	instr
1	-	a
2	x	a
3	x	a
4	x	a
5	y	a
6	y	b

Figure 12a. Trace

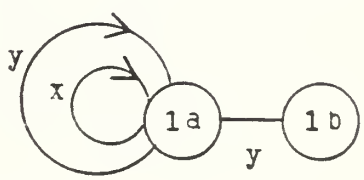


Figure 12b. Program

Figure 12. Minimum State Machine

The best solution lies somewhere between these endpoints. A reasonable first guess at the number of states required to produce a deterministic machine within this spectrum can be made by establishing a lower bound on the number of states. The cardinality of the instruction set is defined as the number of different instructions appearing in the trace. Using the above figure as an example, it can be determined that the cardinality of the instruction set is two; that is, there are two different instructions, 'a' and 'b', in the trace. This measure provides an absolute lower bound on the number of states required in the final machine. This lower bound can be refined by determining a lower bound on the number of states needed for each individual instruction. Once again, using the above figure as an example illustrates this concept. The instruction 'a' at level 5 must be different than the instructions at levels 1 through 4 because of difference set resolution, or else nondeterminism results on the transition 'y'. Therefore, in order to maintain determinism, the instruction 'a' must be allowed at least two states. Summation of the lower bounds for each of the instructions gives a lower bound on the total number of states required for the machine. For this particular example, the program would be generated as:

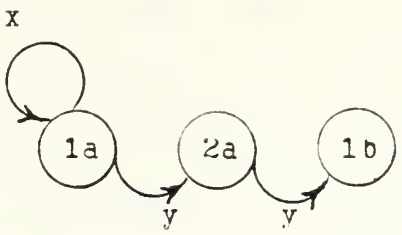


Figure 13. Instruction Set Lower Bounds

If the search space is viewed as a tree structure then the levels of the tree can be associated with the instructions by assigning the first instruction in the input trace to the first level, the second instruction to the second level, and so forth. The branching factor at each level is the state lower bound computed for the instruction seen at that level. The prefix label assigned to the instruction is represented by the specific branch used to traverse to the next level.

The idea of providing a lower bound on the number of states leads to an iteratively expanding depth-first search. When all possible combinations of prefix labels have been tried, but the algorithm remains non-deterministic, the lower bound is incremented and the search is restarted from the top level. When the lower bound is increased, the search tree obtains additional paths to the final solution by increasing the branching factor associated with one or more instructions. The depth of a successful search into the tree

is restricted by the lower bound on the number of nodes required by the deterministic machine. Only when a pattern of prefix assignments has been made which allows the algorithm to remain deterministic and all of the instructions in the original trace have been assigned prefix labels will the synthesis terminate. This mechanism prevents a straight-line model from being output as the algorithm unless it is the only one that can satisfy the input trace. More importantly, it provides the minimum-state deterministic machine capable of executing the input trace.

D. SYNTHESIZER STRUCTURE

The synthesis program is subdivided into two primary modules: static processing of the input trace; and dynamic processing of the information extracted from the input trace by the preprocessing, or static processing phase. Static processing provides information such as couple-classes, difference sets, and lower bounds on the number of machine states. Dynamic processing uses knowledge inherited from preprocessing to guide the search mechanism to a final output of the algorithm. These two modules will be discussed in turn, and the primary mechanisms involved will be amplified.

1. Static Processing

Static processing can be conceptualized as consisting of three main functions: (a) accept the input trace; (b) preprocess the trace for difference sets,

couple-classes, and state bounds; and (c) prepare a trace table for further use by dynamic processing. Once this preprocessing has been accomplished, the static module is no longer necessary to the synthesizer.

In the current configuration, the static module expects to find the input as a sequence of instruction-condition-instruction triples. Figure 14 is an example of an input trace.

level	trace
1	anp
2	psa
3	aga
4	ayr
5	rsr
6	rsr
7	rra
8	aga
9	ayt

Figure 14. Typical Input to Static Processor

Each line consists of a triple, for example 'anp'. The 'a' represents an instruction, the 'n' represents the condition which causes the program trace to transition to the next instruction 'p'. For each level, the first element represents the same instruction as the last element of the preceding level. This is easier to see if the above trace is represented as a Moore machine in which the nodes are instructions and the conditions are transitions. State 'a' transitions on condition 'n' to state 'p' which transitions on condition 's' to state 'a' which transitions on condition 'g' back to state 'a', etc.

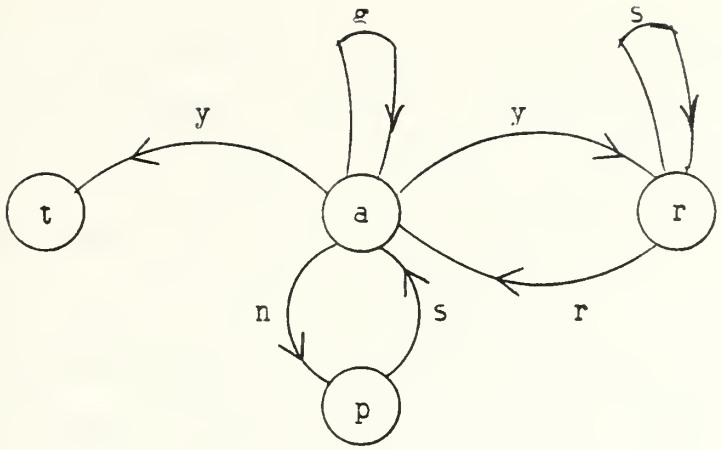


Figure 15. Moore Machine for Input Trace

level	trace	c-c	difference set
1	anp	-	-
2	psa	-	-
3	aga	1	{8}
4	ayr	-	{9}
5	rsr	2	-
6	rsr	2	-
7	rra	-	-
8	aga	1	-
9	ayt	-	-

Figure 16. Intermediate Trace Table

Each occurrence of an instruction symbol in the input trace is represented by the same state at this point in the synthesis.

Once the input trace has been accepted, static processing can begin. Static processing consists of determining the level indices associated with each couple-class and with each difference set. For the trace of Figure 15, these are shown in Figure 16.

There are two couple-classes in this trace. They are [aga] at levels 3 and 8, and [rsr] at levels 5 and 6. The remaining levels are not assigned to a couple-class because no other levels match with them. Couple-class information is useful to the dynamic processor for determining forced assignments and dynamic non-equivalence. These ideas will be discussed more fully in the section on dynamic processing.

Difference sets exist for levels 3 and 4. Level 4 has a difference set which contains the index 9; that is, the element at level 4, 'ayt', must have a different prefix label on 'a' than the element at level 9, 'ayt'. If the 'a' is not labelled differently during the synthesis, nondeterminism will result since the same transition would lead to different nodes.

Difference set resolution is a very powerful mechanism for ensuring deterministic behavior of the algorithm. A considerable amount of the prefix label assignments to the nodes can be resolved using difference

sets. Notice that level 8 appears in the difference set for level 3 even though levels 3 and 8 are in the same couple-class. At first this appears contradictory since equivalent couple-class names imply that the elements are the same, but difference set existence forces the lead instructions to be different. This points out the relative power of couple-class information and difference set information. Difference set information is immutable. Couple-class information only hints at equivalence. In this particular example, the entry at level 3 was caused by the chaining effect of difference set resolution. Notice that since the 'a' at level 4 must be different than the 'a' at level 9, and notice that since the trailing 'a' at level 3 is, by definition, the same as the leading 'a' at level 4, the trailing 'a' at level 3 cannot be the same as the trailing 'a' at level 8; therefore, levels 3 and 8 cannot be in the same couple-class.

To compute the lower bound on the number of states in the algorithm, the minimum number of states needed for each instruction is summed. For this same example, the instruction set consists of {a,p,r,t}. The bounds for p,r, and t are each 1. The bound for 'a' is 2. There must be at least two different occurrences of 'a' from the difference set resolution. Therefore, the minimum number of states with which a deterministic Moore machine can be constructed for this trace is 5.

Finally, static processing passes all the information concerning the input trace to the dynamic processor via a trace table in the following form. Each level has only one associated condition and one associated instruction. Since difference set information is associated with the lead instruction in an instruction-condition-instruction sequence, it is entered at that level. Since couple-class information is associated with the entire instruction-condition-instruction sequence, it is associated with the trailing condition-instruction pair.

level	condition	instruction	c-c	difference set
1	-	a	-	-
2	n	p	-	-
3	s	a	-	{8}
4	g	a	1	{9}
5	y	r	-	-
6	s	r	2	-
7	s	r	2	-
8	r	a	-	-
9	g	a	1	-
10	y	t	-	-

Figure 17. TraceTable

2. Dynamic Processing

Dynamic processing involves assigning prefix labels to the states of the machine. In this way, separate occurrences of the same instruction are differentiated. The dynamic processor is the search mechanism for the synthesizer. It operates in such a way that, at any point in

the synthesis, the portion of the trace previously processed represents a deterministic Moore machine. In order to maintain the determinism, dynamic processing steps through three phases:(1) assignment of the prefix label to the instruction; (2) difference set resolution, and (3) dynamic equivalence assurance. Additionally, each of these phases have built in fixup and backup conditions associated with them. The fixup/backup conditions encountered during difference set resolution or during dynamic equivalence checking are indicators that, if the current assignments remain the same, a nondeterminism will occur in future assignments. As such, they inform the pruning mechanisms of the search algorithm.

An integral part of the dynamic processor is the failure memory. It controls the search. The failure memory may be conceptualized as a L x M matrix where L is the row size and corresponds to the number of levels in the trace. Each row has M columns where M is equal to the lower bound assigned to the instruction contained on that level of the trace. An entry into the failure memory at some level i and some column j, where 1 <= i <= L and 1 <= j <= M, prevents the assignment of j as a prefix label for the instruction at level i. When a failure memory cell contains an entry it is called a valid cell; otherwise it is invalid. Each cell of the failure memory is a two-element entry. The structure factor is the first element. It indicates which level of the

trace caused the entry. The free state factor is the second element. As the name indicates, this element is a function of the number of free states available at the time of assignment. The specifics of the failure memory operation and the nature of failure memory entries will be discussed throughout the rest of the section as each phase of the dynamic processor is discussed.

a. Label Assignment

As previously mentioned, label assignment is the first function provided by the dynamic processor. A label assignment can be either forced or arbitrary. Additionally, the assignment can result in the creation of a new state, a label-name combination not seen before. A forced assignment occurs when the instruction at the current working level is a member of the same couple-class as an instruction at a prior level, and the lead instruction into both of those levels has the same label assignment. The current working level is defined as the level of the trace which contains the most recently assigned prefix label, but difference set resolution and dynamic equivalence checking have not been completed at that level. An example is given in the trace shown in Figure 18.

The label at level 7 is forced by the label assignments at levels 4 and 5. Notice that the instructions at level 5 and at level 7 are in the same couple-class,

level	condition	instruction	c-c	label
.				
.				
.				
4	a	a	-	2
5	n	r	3	1
6	r	a	4	2
7	n	r	3	.1
8	r	a	4	.2
.				
.				
.				

.indicates forced move

Figure 18. Partial Trace Labelling

and that the instructions at levels 4 and 6 have the same prefix label. This condition forces the instruction at level 7 to have the same prefix label as the instruction at level 5. The Moore machine representation of the partial trace is shown in Figure 19. The assignment at level 8 is also forced for similar reasons. By definition, any forced assignment involves previously assigned states, label-instruction combinations, that have been seen before; therefore, no forced assignment can result in a new state.

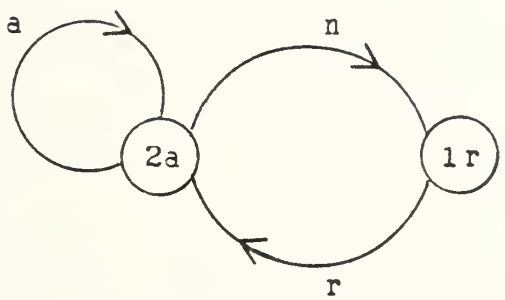


Figure 19. Partially Determined Moore Machine

The failure memory can be used in conjunction with forced assignments to signal a backup condition to the search. If the failure memory entry corresponding to the label assignment at the current working level is valid, then a contradiction results from the forced assignment. Suppose that the trace table and failure memory are as shown in Figure 20, and the forced assignment at level 8 has just been made. The entry '1.1' at row 2, column 8 of the failure memory is interpreted in the following manner. The integer to the left of the decimal indicates that the entry was caused by the current assignment at level 1. The '1' to the right of the decimal point is the number of free states + 1 available when the assignment at level 1 caused the failure memory entry; therefore, when the entry was made there were no free states available. A free state is one which has not been bound to a particular instruction.

The assignment at level 8 is forced. In other words the sequence of the previous assignments causes the prefix label of the instruction at level 8 to be a 2. However, the failure memory contains an entry at row 8 column 2, FM(8,2). This entry indicates that the instruction at level 8 cannot be assigned the label '2', for if it were to be assigned a '2', a nondeterminism will result. To resolve the conflict, backup is initiated until the last unforced assignment is found. In this case, the backup is to level 6.

The assignment at level 6 will be changed and the search will continue from there.

Trace Table					Failure Memory		
level	cond	instr	c-c	label	1	2	3
.							
.							
.							
4	a	a	-	2	-	-	-
5	n	r	3	1	-	-	-
6	r	a	4	2	-	-	-
7	n	r	3	.1	-	-	-
8	r	a	4	.2	-	1.1	-
.							
.							
.							

Figure 20. Trace Table/Failure Memory Configuration for a Forced Assignment

If the assignment is not forced, the failure memory row corresponding to the current working level is searched for the first occurrence of an invalid cell. An invalid cell is one which does not contain a failure memory entry. If a cell is invalid, the assignment of a prefix label corresponding to the failure memory column index for that cell is possible on that level of the trace. The column number of the first invalid cell becomes the label assignment for the instruction at that level. For example, suppose level 6 is the current working level and the trace table and failure memory have the configuration shown in Figure 21.

Trace Table			Failure Memory			
level	cond	instr	1	2	3	4
.						
.						
.						
6	r	a	1.1	4.1	-	-
.						
.						
.						

Figure 21. Trace Table Entry Showing Arbitrary Assignment Method

The first invalid entry in the failure memory on row 6 is in column 3; therefore, instruction 'a' for level 6 will be assigned a prefix label of 3. These non-forced assignments may result in the creation of a new state; that is, a label-instruction pair not previously assigned during the synthesis. If, at some future point in the search, a backup is initiated that reaches this level of the trace, the backup mechanism will not stop to perform a retry. At any point in the synthesis, all previous levels have received assignments based on the constraint that the minimum number of states has been used consistent with maintaining determinism; therefore, assigning a different prefix label to a state which has been defined as a new state only changes the name of the state, and does not change the structure of the algorithm. Since the structure of the algorithm has not been changed, the cause of the nondeterminism is still present.

One other type of assignment should be mentioned at this point. Pseudo-assignment occurs when there is only

one invalid cell left in a failure memory row at a level other than the current working level and there are no free states available. Although pseudo-assignment does not immediately cause a label to be assigned to the instruction at that level, it does simulate a look-ahead mechanism for the search technique by triggering difference set resolution and dynamic equivalence checking as if that level of the trace were assigned a value. Since the pseudo value is the only value currently possible for that level, if a backup or fixup condition is encountered during pseudo assignment, the assignment mechanism can immediately try another label at the current working level; thereby saving the unnecessary search of a path which it already knows to be nonproductive.

Once a tentative label assignment has been made to the instruction at the current working level, difference set resolution and dynamic equivalence checking can be performed. Although these actions may cause a fixup on the prefix label at the current working level, their primary purpose is to furnish information to the failure memory that will help guide future label assignments.

b. Difference Set Resolution

Difference set resolution prevents future assignments being made that are known to cause nondeterminism if the current assignments remain unchanged. Difference sets outline a significant portion of the structure of the input trace without regard to label

assignments in that they prevent nondeterminism from occurring as a result of the same transition out of a state leading to more than one following state. Consider Figure 22.

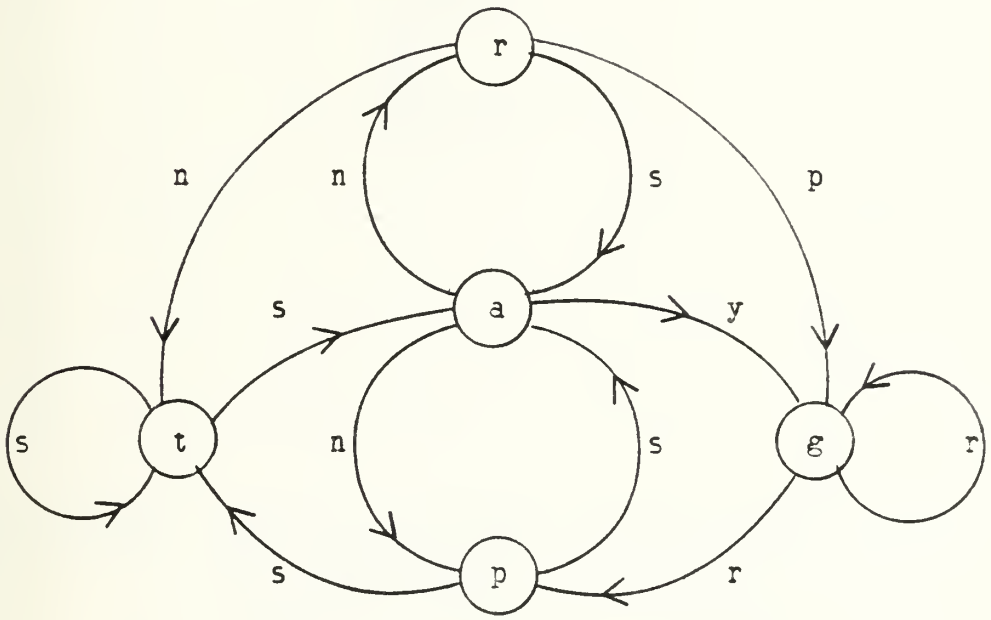


Figure 22. Nondeterministic Input Trace

There are several instances where difference set resolution will force a state to be split into two or more different states. States 'a', 'g', 'p', and 't' all have nondeterministic transitions associated with them. The trace table and failure memory configuration for this trace is shown in Figure 23.

Trace Table					Failure Memory			
level	cond	instr	c-c	difference set	label	1	2	3
1	-	a	-	{3,5,15,18}	1	-	-	-
2	n	p	1	{4,11}	1			
3	s	a	2	{5,15,18}	2	1.1		
4	n	p	1	{11}	2	2.1		
5	s	a	2	-	3	1.1	3.1	
6	n	r	3	-	1			
7	s	a	-	-				
8	y	g	-	{9,10,20}				
9	r	g	4	{10,20}				
10	r	g	4	{20}				
11	r	p	5	{21}		2.1	4.1	
12	s	t	-	{13,14,17}				
13	s	t	6	{14,17}				
14	s	t	6	-				
15	s	a	7	-		1.1	3.1	
16	n	r	3	-				
17	n	t	-	-				
18	s	a	7	-		1.1	3.1	
19	n	r	3	-				
20	p	g	-	-				
21	r	p	5	-				4.1
22	s	a	2	-				

Figure 23. Trace Table/Failure Memory Configuration After Assignment at the Fourth Level

As dynamic processing proceeds with label assignments, difference set resolution occurs. Difference sets are resolved by making an entry into the failure memory row at the level corresponding to the difference set element, and the column corresponding to the prefix label assigned to the instruction at the level from which the difference set is being resolved if the cell has not already been made valid through a previous assignment. For example, if the prefix assignment at level 1 is a '1', the failure memory entries are made in column 1 at levels 3,5,15,18.

Similarly, when the assignment '1' is made at level 2, failure entries are made at levels 4 and 11. Now when the assignment at level 3 is made, the dynamic processor will not try to assign a prefix value of '1' since the failure memory cell at (3,1) is valid. The assignment will automatically be '2'. Notice that at level 5 the previous assignments have caused the prefix label to be a '3'. In other words, the failure memory has caused the search tree to be pruned so that an assignment of '1' or '2' will not be tried. Either one of these assignments would have resulted in nondeterminism being introduced into the trace at level 6.

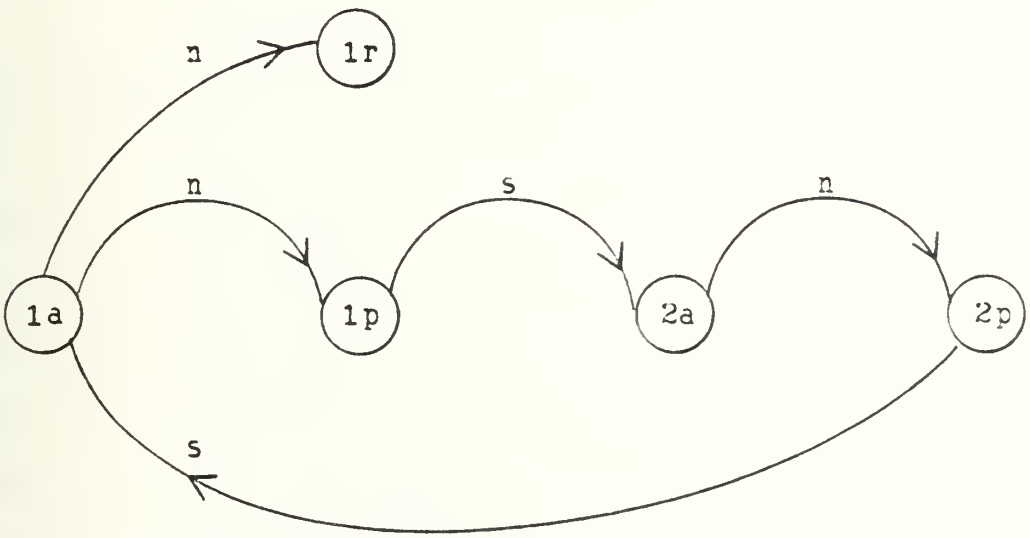


Figure 24a. Prefix Label Equals 1

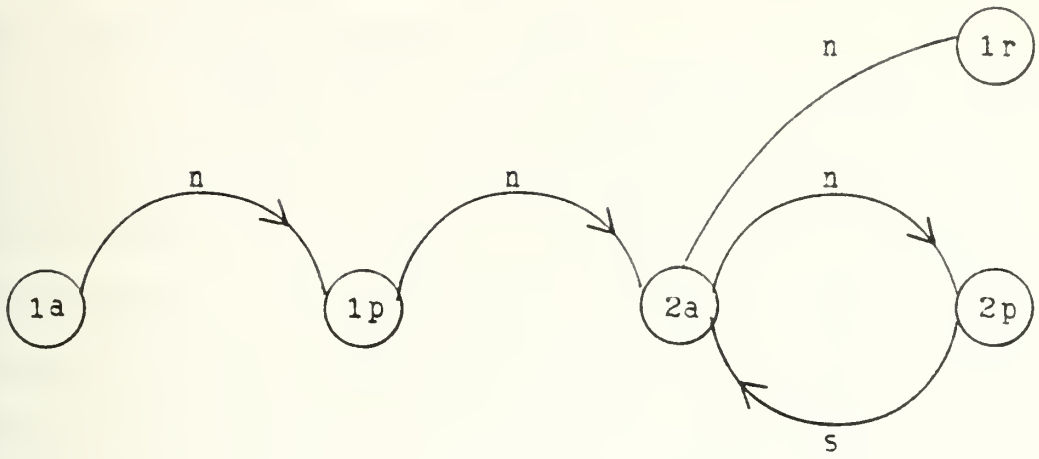


Figure 24b. Prefix Label Equals 2

Figure 24. Nondeterministic Prefix Label Assignments

While failure memory entries are being made under difference set resolution, it is possible for a row to have all cells valid except one. This has been previously defined as a situation leading to pseudo-assignment. This situation has occurred at level 11 in the example given in Figure 23. When such an occurrence happens a look-ahead mechanism is triggered to resolve the difference set at that level. In this example, the failure memory cell at (21,3) has been validated with an entry which indicates the current working level as level 4 when the pseudo-assignment occurred at level 11. Another situation which can occur in a failure memory row is when all the entries in the row become valid.

This condition is called an incipient fence. When an incipient fence exists and there are no free states available, then no assignment can be made at that level. This condition is called a fence.

Since the search mechanism always knows the level from which it is doing look-ahead by difference set resolution, it is able to perform a fixup on the label assignment at the earliest possible time. A fixup is accomplished by incrementing the prefix label by one. If an entire row in the failure memory becomes valid and there are no free states available a fixup must be performed on the label assignment at the current working level. If the label is left the same, then when the search reaches the fenced level, no assignment will be possible. Each time a fixup occurs, all entries made in the failure memory as a result of the previous label assignment are deleted, and entries are then made based on the new label.

c. Dynamic Equivalence

Couple-class information furnished by static processing aids in the determination of dynamic nonequivalence. Dynamic nonequivalence can occur during the synthesis at any level below the current working level when the couple-classes are equal. Dynamic equivalence results when instructions in the same couple-class have been assigned the same prefix label. Consider Figure 25. The I-C-I triples at levels 5 and 6 and at levels 11 and 12 are

[aga]; therefore, they are in the same couple-class. The instruction 'a' at level 5 has been assigned a prefix of '2', and the instruction 'a' at level 6 has been assigned a prefix of '1'. Now, if the instruction at level 11 is assigned a prefix of '2' and the instruction at level 12 is assigned a prefix of '1', dynamic equivalence will occur. Further, the assignment at level 12 will be forced. Dynamic non-equivalence results when such an assignment scheme causes non-determinism. Dynamic equivalence checking functions as a look-ahead mechanism by preventing the future occurrence of a forced assignment which will result in nondeterminism. Suppose the synthesizer is inspecting the trace in Figure 6, and has just assigned the instruction at level 6 a prefix of '1'.

Notice that level 12 is in the same couple-class as level 6. Since the instruction at each of these levels is in the same couple-class, the possibility exists that they may be the same instruction. If the instruction at level 11 is assigned a label of '2' when the working level reaches that part of the trace, then the assignment at level 12 will be a forced assignment of '1'. However, an entry has already been made in the failure memory at (12,1) which indicates that the instruction at level 12 cannot be assigned a prefix label of 1.

	TraceTable				Failure Memory		
level	cond	instr	c-c	label	1	2	3
.							
.							
.							
5	d	a	1	2	4.1	-	-
6	g	a	2	1	-	-	-
7	g	a	2	-	-	-	-
.							
.							
.							
11	f	a	-	-	-	6.1	
12	g	a	2	-	4.1	-	-
13	n	a	3	-	-	-	-
.							
.							
.							

Figure 25. Trace Table/Failure Memory

In order to avoid this contradiction and a backup, dynamic nonequivalence processing causes an entry at (11,2) of the failure memory which corresponds to the labelling of '2' given to the instruction at level 5. Once this is accomplished, when the working level descends to level 11, an assignment of '2' cannot be made and as a result, the assignment at level 12 will no longer be forced by dynamic equivalence which gives the synthesizer a chance to try other assignments that will maintain determinism of the algorithm.

Pseudo-assignment conditions and fixup conditions can occur in the failure memory as a result of validation of all but one of the failure memory cells in a row in the same manner that they occur in difference set resolution. Additionally, dynamic equivalency and difference set resolution can interact to cause failure memory entries

in the following manner. If a failure memory entry is made by difference set resolution at any level which is in the same couple-class as a level previously assigned a prefix label, and if the failure memory entry prevents the assignment that will cause the instructions to become part of the same state, then dynamic nonequivalence will result; therefore, an entry must be made in the failure memory to indicate this condition.

3. Backup/Fixup

The discussion of backup and fixup conditions has been saved until last. The basic idea behind constructing the synthesizer is to provide as much information as possible to the search mechanism, and thereby direct the label assignment with a minimal number of retries. With this in mind backup and fixup become last resorts.

The fixup operation attempts to resolve nondeterminism by incrementing the label at the current working level when a contradiction occurs. If the newly incremented label is not a legal assignment or does not correct the contradiction, then backup must be initiated. The fixup operation cannot be attempted if the assignment at the current working level is forced or if the assignment created a new state. In either of these cases, a fixup operation would leave nondeterminism in the algorithm.

If a fixup fails, or cannot be attempted, backup is initiated. Backup must be initiated from the current working

level when any level is discovered which contains one of these conditions:

1) The label assignment is forced and the failure memory cell corresponding to that level and label is valid.
2) The label assignment causes a contradiction and represents a new state, or
3) There is no free state available for the instruction at a particular level, and all entries in the failure memory row at that level are valid.

The backup begins at the current working level regardless of which level triggered the mechanism, and continues until none of the three conditions given above are present. At that level a fixup operation is attempted and the search begins anew. Any entries into the failure memory which were caused by levels greater than or equal to the new current working level are invalidated by resetting the failure memory entries to (Ø,Ø). Additionally, any assignments are deleted along with their side-effects, such as annotations on forced assignments and new states. If backup causes the working level to be decremented to zero, a free state is added for the use of the first instruction needing more states than initially allotted as the lower bound.

III. PREPROCESSOR

A. PROBLEM SPECIFICATION

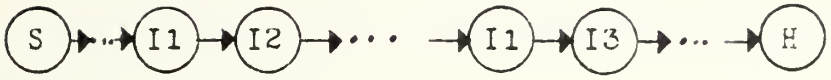
The program synthesizer expects a set of triples where each triple is an instruction, a condition, and an instruction. Biermann [2] has shown that conditions inadvertently or purposely omitted by the user may be inserted into a trace. The algorithm for insertion of conditions collects the set of atoms seen on the transitions for an instruction. An atom is an entity which has a value of either 'true' or 'false'. A condition is composed by logical conjunction and disjunction operations on atoms. For example, an atom may be 'c <= 0', but a condition may be 'c <=0 and a = 4'. A set of minterms is computed from the set of atoms and one of the minterms is inserted after each occurrence of that instruction in the trace. If {a,b} is a set of atoms, then the set of minterms will be {{a,b},{-a,b},{a,-b},{-a,-b}} where - stands for logical negation. It has been shown in reference [16] that only one of the minterms can be true for each occurrence of a transition from any single instruction.

One problem with the algorithm is that it is incapable of inserting conditions if the user has failed to supply any atoms after a particular instruction. For example, if the user should specify instruction I1 followed by instruction

I2 in one part of the trace and instruction I1 followed by I3 in another part of the trace, but the user fails to provide a condition after either occurrence of I1, then the algorithm will be unable to generate a condition for I1. It is assumed that I1 does not appear with an atom elsewhere in the trace. The synthesizer will force two states for I1 to resolve any nondeterminism. This mechanism is fully explained in Section II. If conditions had been supplied in the above example, the difference in the two programs would be the number of states assigned to instruction I1. Figure 26 shows a partial computation without explicitly expressed conditions along with the associated synthesized program fragment. Figure 26 assumes that I1 does not appear elsewhere in the trace. Figure 27 is a representation of the same partial computation except that the conditions c1 and c2 have been explicitly expressed. The computations in both figures are the same, and each program fragment will correctly execute either trace; therefore, the programs must be equivalent programs with respect to program behavior. However the program in Figure 27 is minimal in that it contains fewer states because the user explicitly supplied the conditions.

(S,...,I1,I2,...,I1,I3,...H)

Example Computation

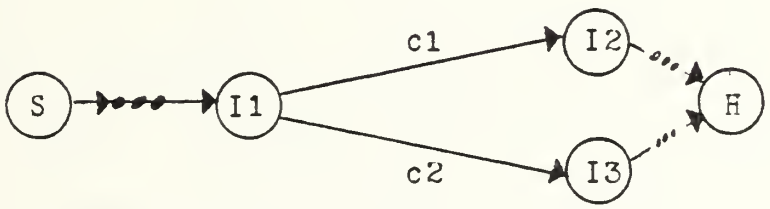


Synthesized Program

Figure 26. Computation without Explicit Conditions

(S,....,I1,c1,I2,...,I1,I3,...,H)

Example Computation



Synthesized Program

Figure 27. Computation with Explicit Conditions

We intend to show that there are mechanisms which can be used to automatically generate the necessary conditions for the correct synthesis of an algorithm produced by an example computation without the user explicitly defining them. The problem may be described as follows. Given an example computation without explicitly defined conditions, infer those conditions necessary to control the flow of computation in a manner such that the synthesized program will demonstrate the behavior desired by the user. In order

to facilitate the solution to the problem, a condition will be viewed as a function that returns a value of 'true' or 'false' when called rather than a logical operation on atomic boolean entities. The problem can then be thought of as constructing a function.

Very little information is available to the current version of the synthesizer when the user provides only a sequence of instructions. Certainly not enough to generate minimal programs as described in Figure 27. This led us to search for other sources of information that would allow us to construct the necessary conditions. We soon realized that the instructions issued by the user do not exist in a vacuum. These instructions manipulate data. If the entire computer memory, including registers, is viewed as the domain of interest, then execution of an instruction always changes this state. Intuitively, the domain also reflects the reason that the user decided to execute a particular instruction. A search of a space of this size in order to determine the reason is impractical; however, observing only those data elements affected by the sequence of instructions can often be quite practical and can significantly reduce the search space.

We chose the text editing domain as the domain of interest since we felt that it would be sufficiently interesting to warrant application of synthesis techniques. This domain was selected because, first, techniques

developed in this domain may be general enough for extension into other domains, secondly, the world for this domain can be described as the set of all characters contained in a particular text file which makes the world finite, and finally, the instruction set is small enough to be manageable.

Although our primary research is directed toward studying techniques to apply to automatic condition generation, we feel that the synthesizer could be a powerful text editor and could provide some useful features not normally seen in conventional text editors. Extended features could include the ability to capitalize the first letter of every sentence, the ability to capitalize all small letters in the text, the ability to identify a string and perform some operation before, after or on it , or any combination of these editing actions.

The working hypothesis is to have the user process the text file in a normal manner and have the synthesizer infer a program from his actions. Two requirements were levied upon the user. The first requirement on the user is that he must inform the synthesizer when he desires to have a program generated so that the synthesizer can begin monitoring the user's actions. A great deal of time was spent trying to figure out methods that allowed one general mechanism to be used to monitor the user's actions and the resulting changes in the text file. Since we could not

produce such a mechanism, a second requirement was levied on the user. This requirement recognizes a basic distinction between two different aspects of text editing: context free substitutions, and context sensitive substitutions. We define a context free environment to be one in which the character to be operated upon is not dependent on characters around it. Capitalizing all occurrences of small letters is an example of a context free operation. A context sensitive operation is defined as an operation in which the action to be performed on a character or sequence of characters depends upon other characters around the main character of interest. Capitalizing the first letter of every sentence is a context sensitive operation. Condition inference in a context sensitive environment is inherently more difficult than in a context free environment in that the condition must be constructed from events which require a look-ahead capability not inherent in the synthesizer. The user will be free to switch from environment to environment at his convenience. The synthesizer will create program segments from each environment which can be used to construct a complete program by a post-processor.

B. DESIGN FOR A CONTEXT FREE ENVIRONMENT

1. Overview

Programs that operate on a single entity can be constructed by the synthesizer. Figure 28 shows the construction of a program from a trace intended to

communicate that the letter "d" should be capitalized wherever it appears in the text file. The column labelled 'trace' contains triples of the form instruction, condition, instruction. B is the start instruction, R is the move right instruction, C is the capitalize or change instruction and S is the stop instruction, respectively. The conditions for this trace are the characters seen in the text file prior to the execution of the second instruction in each triple. The special condition "Ø" is the null condition, and is always inserted after the start instruction.

The generated program will correctly execute the trace that was used to construct it, and by examination of the program it can be shown that the program will convert all d's to D's in a text file consisting of the characters A, b, C, d, F and G. There are no arcs available for other characters in the character set. In order to generate a program to perform the same function on an arbitrary text file, the user would be forced to give an example of the desired transition for every character in the character set.

Since it is desirable to relieve the user of the chore of providing an inordinate number of examples in order to completely specify the function, a method is required that utilizes a few examples of the types of conditions that are to appear on the arcs to generalize the conditions into a more compact and complete form. If a generalization can be found, the multiple arcs may be replaced with a more general

condition and, therefore, correct programs can be created with fewer examples. However the combination of arcs between nodes must be accomplished so that determinism is maintained or the synthesizer will not create a mimimum state machine capable of performing the desired function. That means that the generalization technique must be able to handle conflicts properly. The arcs in Figure 28 that originate at state R and terminate at state R appear to consist of elements from the capital letters and small letters. The generalization of {x| x ∈ capital letters} U {z| z ∈ small letters} would appear to be a reasonable replacement for all of the R to R arcs. If this generalization was made a conflict would result because the letter 'd' is also an element of the {z| z ∈ small letters}.

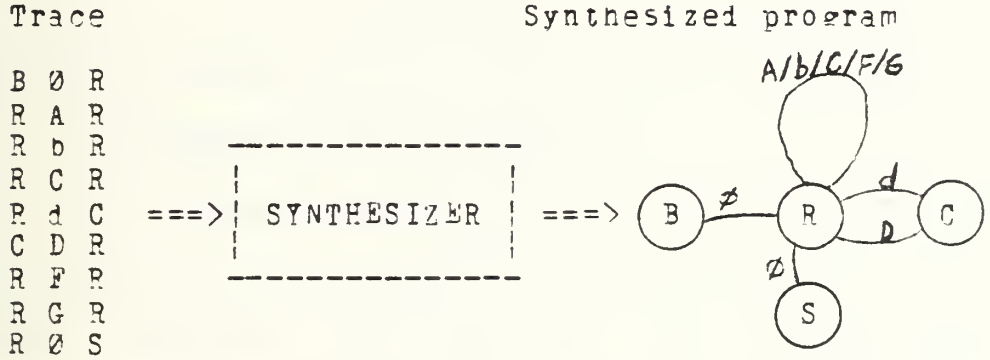


Figure 28. Synthesizer Action

2. Structure of the Condition Preprocessor

The preprocessor is designed to accumulate knowledge from the traces it is provided, then use the knowledge to construct meaningful conditions. The preprocessor scans the input trace looking at the instructions and characters that

are seen before the instructions. This phase extracts pairs of instructions from the trace. The trace in Figure 28 would have the instruction pairs (B,R), (R,R), (R,C) and (C,B) extracted. Attached to each of these pairs is the set of characters that were seen between the pair. The preprocessor then analyzes the information to determine if a generalization can be made from the set of characters associated with each instruction pair.

The natural division mentioned above allows the preprocessor to be divided into two modules. The first module performs the scanning function while the second module analyzes the information and applies a heuristic to provide the most general condition possible. The implementation of the preprocessor will be discussed later, but before it can be discussed an explanation of the data structures required by the preprocessor is needed.

3. Preprocessor Data Structures

To simplify the problem we define two types of instructions in this domain. Instructions that specify the current location of interest are cursor positioning instructions. Instructions that change the state of the domain are data manipulation instructions. The preprocessor accepts as input a sequence of instructions and an associated sequence of characters. The first instruction in the instruction sequence is always the start instruction which does not have a character associated with it. The last

instruction in the sequence is always a halt instruction. Every action performed by the user is captured and appended to the instruction sequence list. The character sequence is created in harmony with the instruction sequence. In the quiescent state the cursor will indicate a certain position in the text. When the user performs some action such as move the cursor right, a monitor picks up the value in the old position and associates that value with the instruction executed by the user. For example, assume a user has a text file in lower case letters that he wants to change to all upper case letters. The user initiates the synthesizer then proceeds across the line of text changing lower case letters to upper case letters. For the purpose of this example, assume the line of text is "change lower case to upper case". As the user moves across the line making substitutions, the condition monitor captures the actions performed and the characters seen. The example line would yield an instruction sequence of (R, C, R, C, R, C, R, C, ..., C, S). The associated character sequence would be; (c, C, h, H, a, A, ..., e, Ø). The "C" and "R" in the instruction sequence are the capitalize and move right instruction, respectively. Note that the capitalize instruction does not reposition the cursor and when the user moves the cursor to the right, the result of the capitalize instruction is associated with the move.

Another data structure needed by the preprocessor is the ASCII vector. The ASCII vector is a 128-byte linear array with indices numbered 0 through 127. Each byte in the array is referenced by the decimal value of a particular ASCII character. For example, the array element reserved for the ASCII character '0' is indexed by 48 decimal. The array element reserved for the ASCII character 'a' is indexed by 66 decimal. The vector defines a partition of the ASCII character set by using the following technique. The ASCII character set has been divided into eight mutually exclusive subsets.

Subset 0	Capital letters
Subset 1	Small letters
Subset 2	Numbers
Subset 3	space character <sp>
Subset 4	Symbols
Subset 5	Punctuation
Subset 6	Arithmetic operators
Subset 7	Control characters

The subset name is entered into the ASCII vector at each cell by converting the ASCII character to its decimal equivalent and using that value as the array index. The default partition is shown in Figure 29.

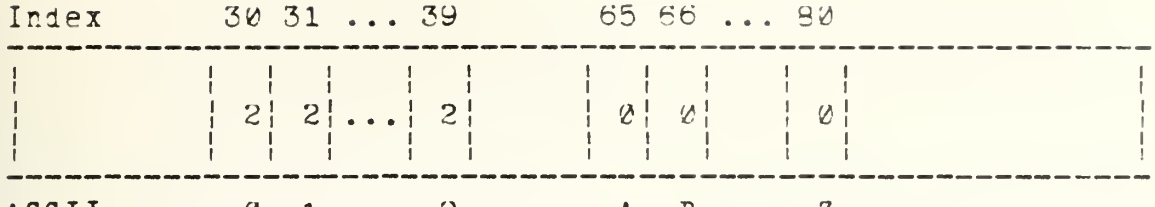


Figure 29. ASCII Vector

The character set hierarchy is defined by the tree structure in Figure 30. The tree is related to the ASCII vector through the character subset names contained on each node one level above the leaf nodes. For the default hierarchy shown in Figure 30, a zero would be entered in the ASCII vector for all capital letters, and a 1 would be entered for all small letters. If a different partition of the character set is required the user can modify the hierarchy or create his own. An example will be given to explain how the modification may be accomplished. Assume a partition is desired where the vowels are isolated into a set. Assume further that the the vowels are to be subdivided into capital vowels and small vowels. The hierarchy would be modified by placing a son called 'vowels' on the alphabetic node. Attach to the new node two sons, called 'Cap-vowels' and 'Small-vowels', with arcs to the appropriate characters. Relabel the hierarchy so that sibling relations are numbered in increasing order. Finally, initialize the ASCII vector with the new labelling. All of the modifications can be done by the system when the user calls for the modification The modified hierarchy is shown in Figure 31.

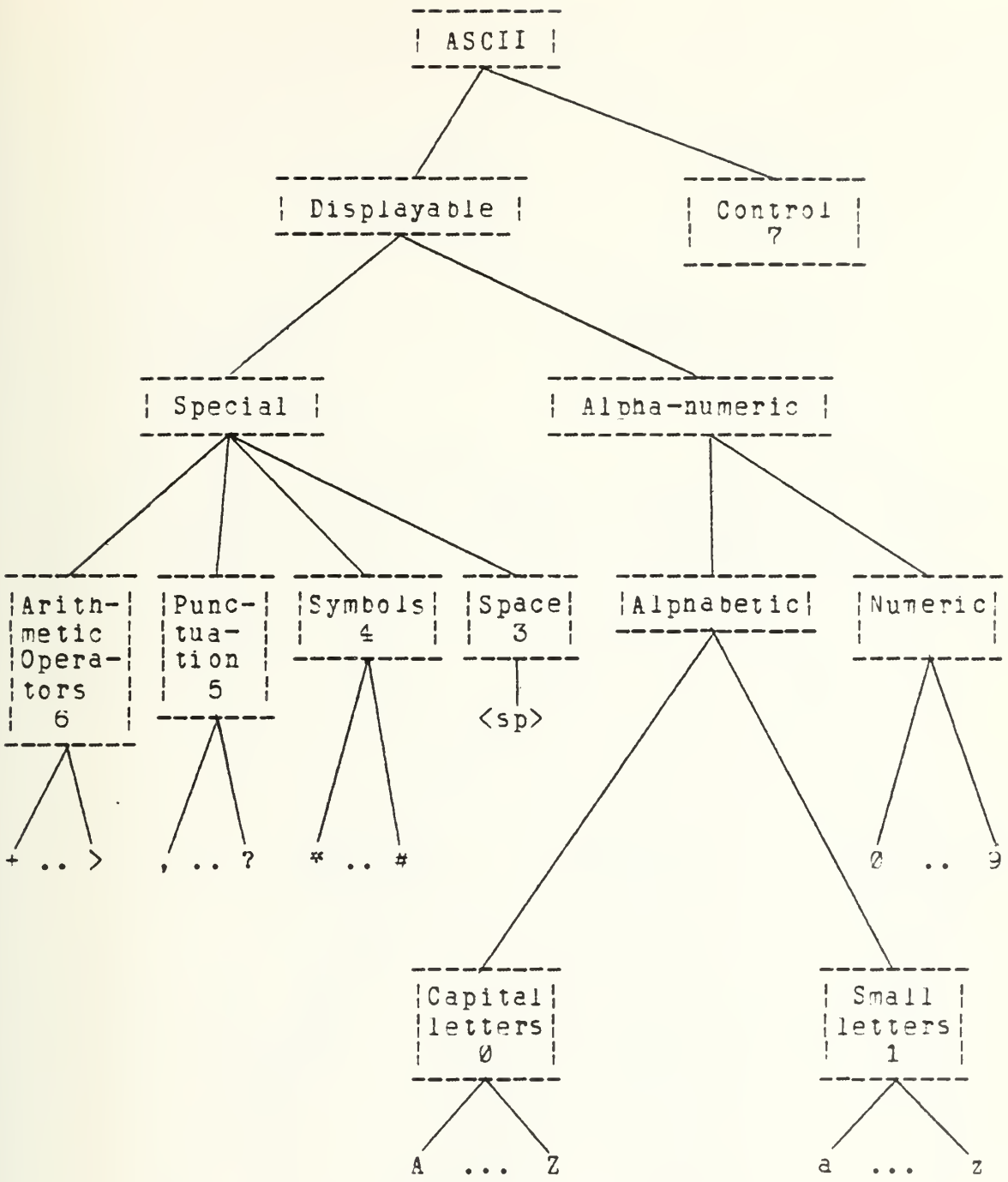


Figure 30. Default Hierarchy

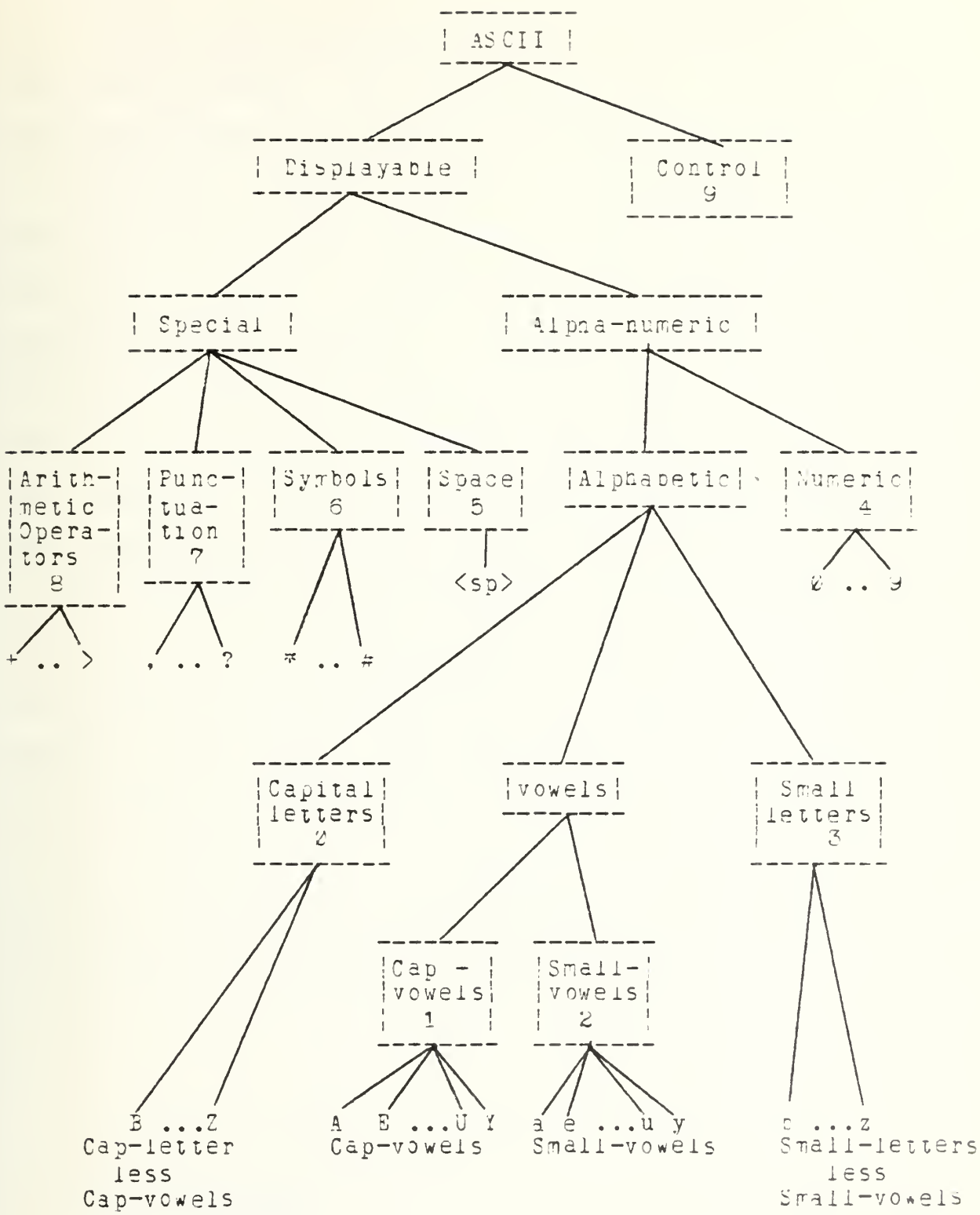


Figure 31. Modified Hierarchy

The next data structure used by the preprocessor is the transition table. The transition table contains the knowledge gleaned from scanning the instruction sequence and the character sequence created by the monitor. Figure 32 shows the format of the transition table. The transition table is an array of records with each record containing information on a transition. In the table, I1 and I2 are instructions where I2 directly follows I1 in at least one place in the instruction sequence. 'Active-sets' is a field that contains information on sets of characters that have been observed by the monitor on the transition from I1 to I2. The fields 'Set-1' through 'Set-n' contain the value for set name, the count of the elements from the set associated with the transition and a pointer to a linked list of the elements. The records that would be created for the trace given in Figure 28 would be associated with the transitions B to R, R to R, R to C, C to R and R to S.

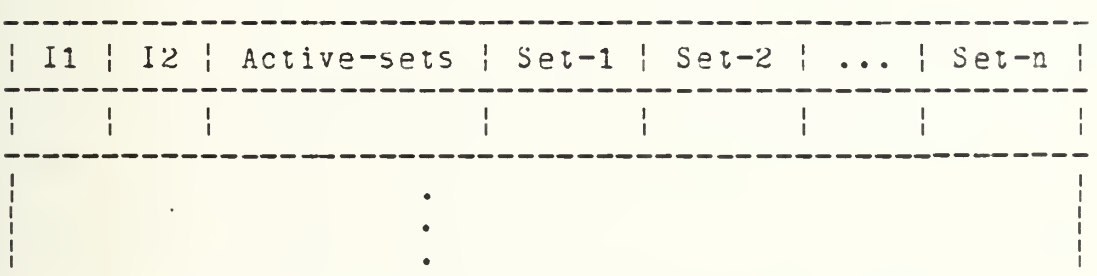


Figure 32. Format of the Transition Table

4. Implementation

The context free preprocessor consist of two main modules; the scanner and the insertion modules. Another important module not part of the preprocessor is the user

monitor. The monitor gatners tne actions of tne user and creates two arrays. One array contains tne sequence of instructions tne user provided and tne otner contains information of what was true before an instruction was executed. Tne information tnat is gatnered is tnen passed to tne appropriate preprocessor.

Tne example instruction and character sequences given in Figure 33 will be the example used to explain tne mechanism of tne preprocessor. Figure 33 is illustrative of a collection of actions tnat were performed by some user. The user's goal is: Change all lower case letters in a text file into upper case letters. The user has activated tne condition monitor, positioned tne cursor at tne beginning of a line of text and moved right along tne line, changing tne lower case letters to upper case wnenever one appeared above the cursor. Figure 33 is an example of output from tne monitor assuming tne line tne user processed was "The numbers 1, 2, 3, 5, 7 ARE prime.". Tne first column in Figure 33 is tne character array. It contains tne character under tne cursor prior to execution of tne instruction in column two. Column two is a trace of tne actions performed by the user. Tne "R" represents tne "move cursor right" instruction and tne "C" represents a cnange without cursor reposition instruction. Figure 33 can be read as: The character in column one was observed and tne instruction in column two was executed.

index	character vector	instruction vector
1	T	R
2	n	C
3	H	R
4	e	C
5	E	R
6	<sp>	R
7	n	C
.	.	.
.	.	.
.	.	.
22	1	R
23	,	R
24	<sp>	R
25	2	R
.	.	.
.	.	.
.	.	.
36	A	R
37	R	R
38	E	R
.	.	.
.	.	.
.	.	.
48	e	C
49	E	R

Figure 33. Monitor Output

The scan module of the preprocessor is activated when the user indicates the representative example is complete. Let 'inst-index' be an index for the instruction array that is initialized to 1. The first step is to create a transition from the start instruction to the first instruction in the instruction array and add the transition to the transition table. This transition will indicate the beginning of the program and will transition to the first instruction provided on a null condition. The module then moves down the instruction array creating other transitions and adding them to the transition table. Duplicate

transitions will not appear in the table. A transition is defined as a pair (I1,I2), I1 and I2 are instructions and I2 follows I1 within the instruction array. The instruction array in Figure 33 yields transitions (R,C), (C,R), (R,R).

The transitions are constructed by indexing through the instruction array. The instruction at inst-index and inst-index + 1 form a transition. The transition is the match against the transition table. If a match occurs, the character in the character array at inst-index + 1 is extracted and its ASCII value is used to index into the ASCII vector. The value stored in the ASCII vector is used as an exponent for two and stored in a temporary variable. A bit by bit logical OR is performed between the temporary variable and the Active-sets variable for the transition and the result is stored in Active-sets. Active-sets contains the information of every set from the partition that has elements seen on the transition. The operation described above allocates one bit for each set in the partition. If Active-sets equals 1 then bit one of Active-sets is a 1 signifying at least one element of set 1 has been seen on this transition. A two would signify that some element of set two had been seen and a three would signify that some element of set one and some element of set two had been seen.

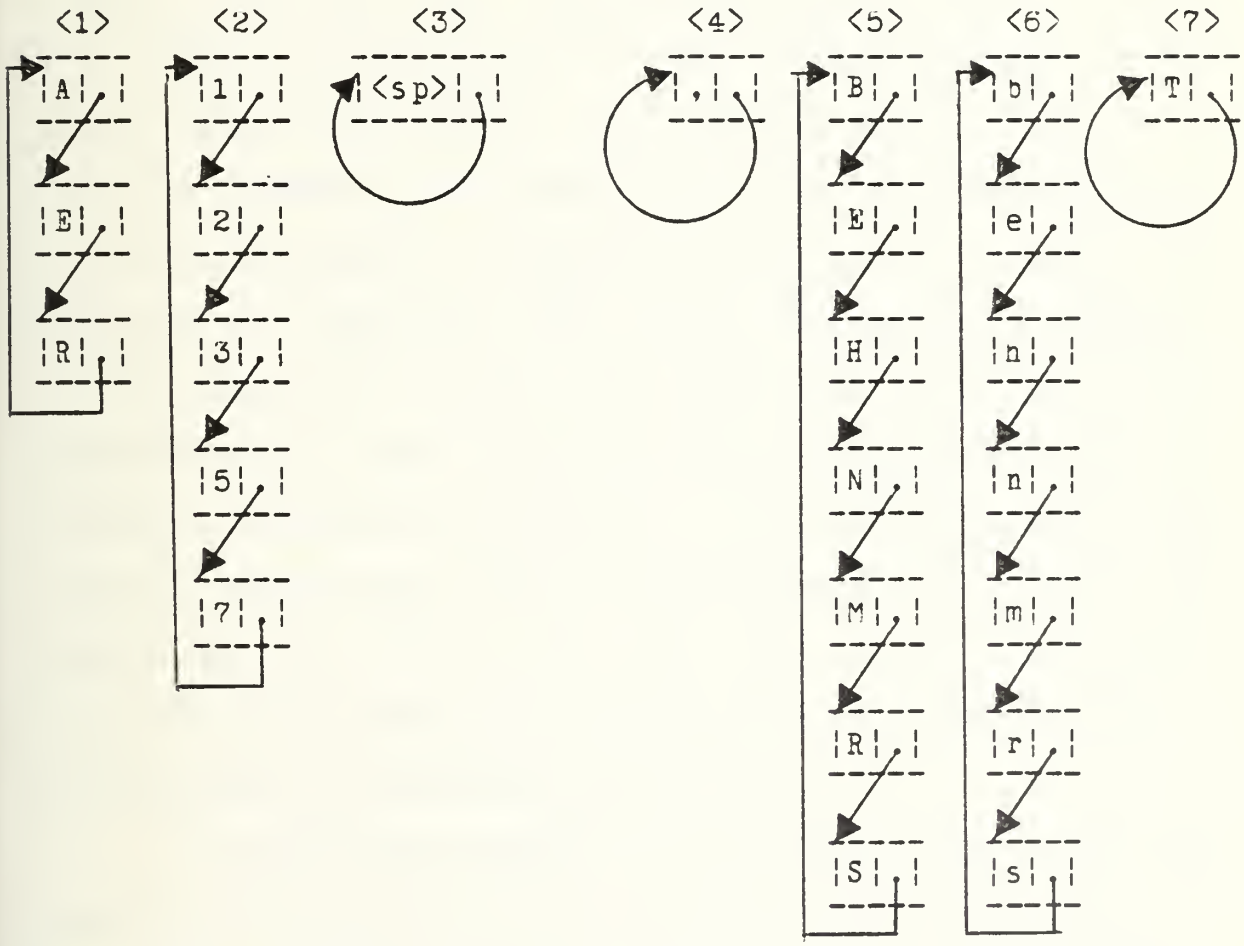
In the transition table are fields for each set that has been determined to be active for the transition. Within

each of the set fields there are three subfields, the first is the set name, the second is a count of the elements seen for the set and the last is a pointer to the start of a circularly linked list containing the elements used from the set. The value that was obtained from the ASCII vector is used as a set name and matched against each of the set fields' set name. If the set name matches an entry the character at inst-index + 1 is added to the linked list in lexicographical order if not already on the list and the count is incremented by one. If a match does not occur on the set name a new set field is created and given the name that was obtained from the ASCII vector, the count is set to one, and the character is put on the list.

When the scan module reaches the end of the input, the transition table contains an entry for each transition that was seen. Each transition is associated with all the sets that had elements seen with the transition. Finally each transition is associated with the actual elements through the linked list for each set. The information is then passed to the insertion module for analysis. Figure 34 shows the completed transition table and the linked list of elements for each set.

Once a completed transition table has been created, control is passed to the insertion module. The insertion module processes the information in the transition table and assigns a condition for each transition.

In1	In2	Active	Set-n	...
B	R	1	0 1 <7>	
R	C	2	1 8 <6>	
C	R	1	0 8 <5>	
R	R	45	0 3 <1>	2 5 <2>	3 1 <3>	5 1 <4>



NOTE: The notation <1>, <2>, etc. represents a pointer to the linked list headed by the same symbol.

Figure 34. Completed Transition Table

The Active-sets entries provide an efficient mecnanism for recognizing potential conflicts on emanating arcs. Performing a bit by bit AND on the Active-sets entries that have a common originating intruction yields the source of conflicts. The bit positions that are on (bit equals 1) are the set (or sets) that have had elements on multiple transitions. For example, let (I1,I2) and (I1,I3) be entries in the transition table with Active-sets value of five (0101 binary) and three (0011 binary) respectively. Let Q equal the result of the bit by bit AND of the Active-sets values given above (i.e. 0001). Q indicates that there is a conflict between the transition (I1,I2) and the transition (I1,I3). Furthermore, Q indicates that the set causing the conflict is labelled zero in the nierarchy of Figure 30 because the on bit is in the right most position which corresponds to two raised to the zero exponent. Using the exponent to enter the hierarchy, it can be determined that capital letters were seen on both transitions. Once all the conflicts for transitions with the same originating intruction are known, the conflicts must be resolved before an assignment of conditions can be made.

Extending the example given above, assume that eight capital letters were seen on transition (I1,I2) and four capital letters were seen on the transition (I1,I3). A partial condition can be constructed for the transition (I1,I2) as a set difference between the set of capital

letters and the actual elements seen on the transition (I1,I3). The partial condition for the (I1,I3) transition becomes the set of capital letters that were actually seen with this transition. The initial conditions for these transitions become the union of the sets indicated in Active-sets as not being in conflict and the sets created by the resolution of conflicts. Therefore, the condition for (I1,I2) is ({ x | x $\in$ capital letters} - {x|x $\in$ capital letters on other transitions}) U {x|x $\in$ numeric}, and the condition for (I1,I3) becomes { z | z $\in$ ({actual capital letters seen} U {small letters})}. In this example, it was assumed that the sets, numeric and small letters, were an appropriate generalization for the transition. In practice it cannot be done without consideration of the number of elements that have been seen from the set on the transition. If the count field for the set exceeds a threshold value for the set, the generalization may be made, otherwise the elements themselves become the partial condition for the transition.

After a condition has been constructed for a transition, a final strong generalization technique is employed. The Active-sets value for the transition again supplies the starting point for this technique. Notice adjacent bits in Active-sets correspond to adjacent nodes in the hierarchy. Therefore, a check is made of the Active-sets to see if it has adjacent bits with a value of one. If it

does then a generalization may be attempted. Assume the condition (({capital letters} - {A E I O U}) U {small letters} U {numeric}) has been constructed for some transition. The Active-sets value for this transition must be seven (0111 binary). With the default hierarchy in Figure 30, a generalization to Alphabetic and then to Alpha-numeric would be attempted. Notice that a generalization to Alpha-numeric would fail because of a conflict with another transition. Intuitively ({alpha-numeric} - {A, E, I, O, U}) would be a correct choice for the condition for this transition. A general procedure for the construction of generalized conditions is given below.

A set of nodes $Y = \{y_1, y_2, \ldots, y_n\}$ is generalizable to a node X if the set of node Y form a complete and exhaustive set of leaves to the subtree rooted at X. Further, a set of nodes $Z = \{z_1, z_2, \ldots, z_m\}$ is generalizable to the set $W = \{w_1, w_2, \ldots, w_j\}$, $j < m$, where each w is a generalization of a subset Z.

IF the condition = $F_1 \cup F_2 \cup \ldots \cup F_n$
where $F_i = z_i - q_i$, $i = 1,n$
where $q_i \subset z_i$ (q_i possibly null)

THEN
the condition is set to $W - \bigcup_{1 \le i \le n} q_i$

where W is the smallest set
$W = \{w_1, w_2, \ldots, w_j\}$
such that W generalizes $\{z_1, z_2, \ldots, z_n\}$

C. DESIGN FOR A CONTEXT SENSITIVE ENVIRONMENT

1. Overview

Condition generation in the context sensitive environment is a more difficult task than in the context free environment. This difficulty arises from the scope of knowledge required to make decisions on what a condition is to be. The conditions themselves are more complex because they depend not only on the character that is being seen, but also depend on characters that precede and follow the current character under consideration. The following example will be used to illustrate the difficulties and our solution to this problem. Assume a user wishes to capitalize all occurrences of the word 'time' in some text file. Also assume that the word occurs at the beginning, at the end, and in the middle of sentences in the text file. The question is how to construct a program that performs the desired function given only the actions the user performs as an example of the required program.

The assumption about the position of the word 'time' in the text file implies that the requested action needs to be accomplished on strings that have very different characteristics. Certainly, both 'time' and 'Time' should be capitalized as should 'time,' , 'time?' and 'time<sp>'. On the other hand the string 'time' should not be capitalized when it occurs within a word like 'sometime' or 'timely'.

Any generated program that behaves as described above must be able to recognize an occurrence of the string or some variation of the string. The totality of this information must be glued together to provide a single condition that is descriptive of what the surrounding environment must be like before the action is performed. The implication is that the condition itself must be able to perform checking and look-ahead. In other words, the condition for the transition to the operation must in fact be a procedure which responds 'true' whenever the string of interest is recognized. Assume for the present that the string of interest can be discerned from the user's actions, (a hard problem by itself, see Angluin [19]) one must wonder how such a procedure can be constructed and then inserted into the generated program which performs the function of a condition on some transition in the program. Figure 35 shows a procedure which recognizes the word 'time'. Note the robustness of the procedure in that it distinguishes between the differing occurrences of 'time' as mentioned above. Figure 35 points out that the problem is not just generating a procedure as a condition but also generating conditions within the procedure that is to be the overall condition. The arcs labeled 'T v t' and '<SP> v {punctuation}' should be noted with interest because they provide the robustness the condition procedure needs. The discovery of arc labels for the condition procedure will be discussed next.

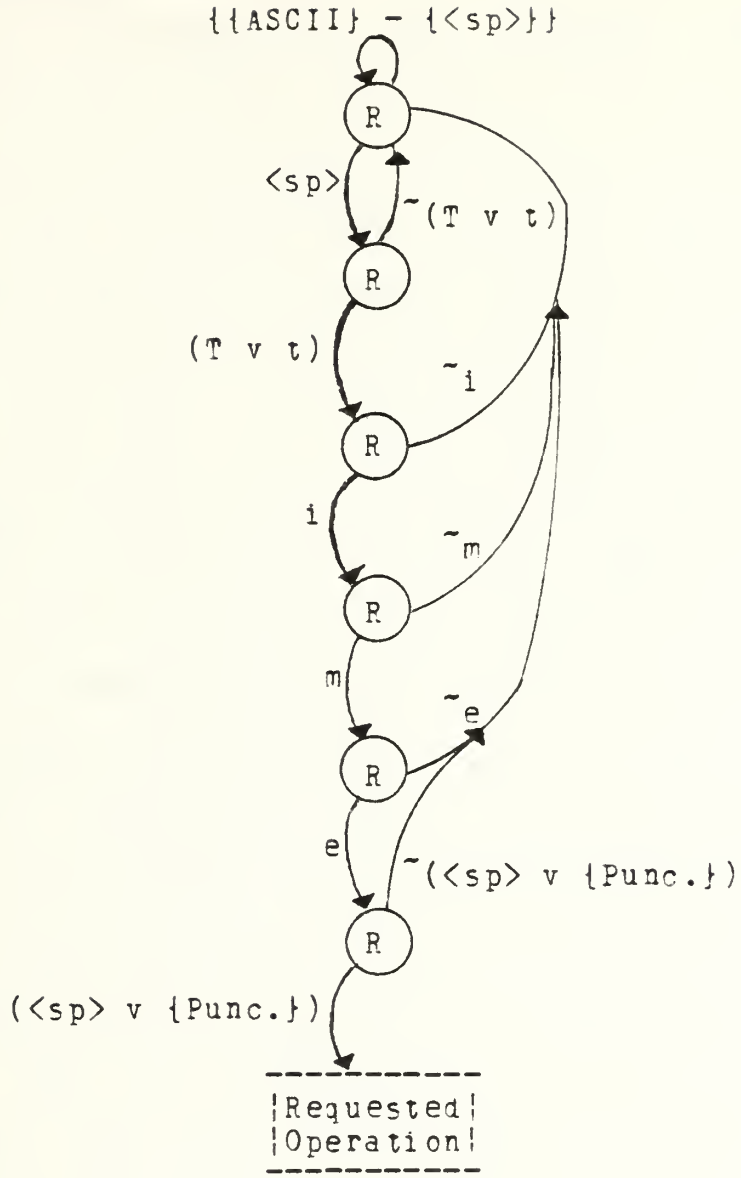


Figure 35. Condition for "time" and "Time".

2. Implementation

The monitoring of user actions provides the instruction and character sequence in the same manner as done in the context free mode. A consideration was given to require more information be provided by the monitor, however, the notion was discarded because it would require the user to be aware of the functioning of the preprocessor. Requiring the user to provide information to the system would betray our goal for the system. The user should only be required to initiate the system and then perform editing as if the system was not actively monitoring his actions. We feel the requirement of specifying whether the user wants to perform context free or context sensitive operations is the maximum that should be asked. If it were feasible to recognize the difference between the two modes from the user's actions alone, this limitation would be also removed.

Given only the instruction sequence, the character sequence, and the information of a context sensitive environment, the first assignment of the context sensitive preprocessor is to discern the string of characters upon which some operation is to be performed. This is a pattern recognition problem of considerable difficulty. Angluin [19] provides the following theorem, "There is an effective procedure which, when given a sample S as input, outputs a pattern p which is descriptive of S.". The sample S is a subset of the set of all strings over the alphabet of the

language. The effective procedure is computationally expensive and not implementationally desirable for our system. The procedure is an enumeration technique on patterns with a length less than the shortest example in the sample set S. Each of the enumerated patterns is tested to determine if it is descriptive of the entire set S. The longest pattern that is descriptive of S is the most specific pattern for the set. Clearly, as the length of the of the sample grows, the number of enumerated patterns will grow exponentially. Angluin [19] states, "In the general case, the test performed on the patterns is an NP-complete problem.". The test she is referring to is the check to see if the enumerated pattern is descriptive of S.

For implementation purposes, we need a mechanism that falls well short of the exponential behavior of the effective procedure mentioned above. The text editing domain has two types of instructions for the purpose of this paper. The first type of instruction will be called cursor positioning instructions while the second type will be called data manipulating instructions. Assuming the text file is to be represented as a linear array, only one cursor position instruction need concern us. All cursor positioning commands such as move left, move up or move down can be represented as move right instructions. Data manipulation instructions operate on one character and do not reposition the cursor.

The method we have adopted for determining the string of interest and the context of the string is based on the above definition of the types of instructions available in the text editing domain. The preprocessor scans the instruction sequence looking for an occurrence of a data manipulation instruction. The character associated with this instruction is then taken as the first character of the string of interest. Other characters are added to the string by continuing the scan until multiple occurrences of cursor positioning instructions are encountered. A hypothesis is then constructed consisting of three parts. The first part is the beginning context. It is constructed from the characters that preceded the string in the character sequence. The second part is the string itself and the final part is the ending context constructed from the characters seen after the string. For engineering considerations, the number of characters in the beginning and ending context will be limited to twenty characters. The probability of the context exceeding twenty characters on both sides of the string in the text editing domain is small enough to ignore.

Once a hypothesis is proposed it is set aside as an active hypothesis and scanning of the input continues. Other cases of data manipulation instructions surrounded by cursor positioning instructions will result in other hypothesis being constructed. As these hypothesis are added to the active hypothesis list they are checked for consistency and

if the new hypothesis causes conflicts they are resolved by constructing anotner nypotnesis from the conflicting hypothesis. To demonstrate this mechanism we present an example which will illustrate the generation of hypotneses and resolution into a condition function. The example used is the construction of the function which will recognize the string 'time'.

Suppose the text file contained the following sentences somewnere in the file.

The time is two oclock.
It is time to go to bed.
Time the runner.
Did you run out of time?

Also, suppose the user nas specified the environment is to be context sensitive and nas begun to perform actions on the file. The monitor could create the following instruction and character sequence fragments from the user moving tnrougn the text file and capitalizing these occurrences of 'time'.

(RRRRCRCRCRCRRRR ...)
(The tTiImMeE is ...)

(RRRRRRCRCRCRCRRRR ...)
(It is tTiImMeE to ...)

(RCRCRCRRRRR ...)
(TiImMeE tne ...)

(... RRRRRRRRRRRCRCRCRCRR)
(... run out of tTiImMeE?)

This example is not to imply the user must change all occurrences in the text file but he should provide enough examples from the file to insure his desires are understood. If the user has not supplied a distinguishing set of examples and an incorrect program is generated he may add to the set of examples.

Scanning the first instruction sequence until the first data manipulation instruction results in the string 'time' being constructed. The resulting hypothesis is that the string 'time' is within the context of 'The<sp>' and '<sp> is two oclock.'. The hypothesis may be viewed as the following data structure.

```
Hypothesis 1:
        Begin context: The<sp>
              String: time
         End context: <sp>is two oclock.
```

A second hypothesis would be generated for the next portion of the instruction sequence as shown below.

```
Hypothesis 2:
        Begin context: It is<sp>
        String: time
        End context: <sp>to go to bed.
```

A comparison of these two hypotheses indicates a disagreement between the contexts. The conflict is resolved by determining the longest beginning and ending context that agree between the two hypotheses and generate a hypothesis reflective of this agreement. By working backward from the last character in the begin context for both hypotheses, it is possible to ascertain that the only character in

agreement is the space. Working forward from the first character in the end context for both hypotheses, again only character in agreement is the the space. A third hypothesis with the new begin and end contexts is generated as follows:

Hypothesis 3:
Begin context: <sp>
String: time
End context: <sp>

This hypothesis specifies that the string 'time' must be preceded and followed by a space. Note the test of the hypothesis implies the user is allowed to specify one string during an example computation. It is also implied that there must be a begin and an end context for the string. Since it is possible to have two hypotheses where one of the context strings do not agree in any of the characters, a method must exist to provide the appropriate context.

Whenever the comparison between context of two hypotheses results in the null string, a disjunction is formed from the characters immediately next to the string. For example, the instruction sequence given above would give the hypothesis:

Hypothesis 4:
Begin context: Did you run out of<sp>
String: time
End context: ?

A comparison between hypothesis 3 and hypothesis 4 would result in the null string for the end context. Since there must be an end context, the disjuction of <sp> and ?

is formed and this become the end context for the new hypothesis. Generalization techniques that were mentioned in the section on context free environment are then applied in an attempt to reduce the end context to the most general context consistent with the data seen. The only alteration in the generalization scheme is the lowering of the threshold values for important sets. In this example, the threshold value for the punctuation set would be lowered to 1 and the ending context would become { x| x=space or x ϵ {Punctuation}}.

The final problem to be solved is the recognition of variations in a string. Examples of variations of a string are, 'Time' and 'time', or 'enclosure' and 'inclosure'. As mentioned, if the user intends to capitalize all occurrences of 'time', 'Time' is to be included. Note these variations of the string become the compound labels for the arcs in Figure 35. The system includes a rule that enables the recognition of variations of strings provided the user gives an example of the variation. The rule simply states that the string length will be established to be as long as the longest string encountered during processing. Again, using the example, the hypothesis for 'Time the runner.' would be:

Hypothesis 5:
Begin context: ... T
String: ime
End context: <sp>the runner.

It has been established by preceding user actions that the string length for the hypothesis should be 4. By

matching the pattern in hypothesis 5 with the string from hypothesis 4 it can be determined that the string in Hypothesis 5 should be expanded by inserting a 'T' in front of the string. Another hypothesis is then generated where the string will be the disjuction between the strings 'time' and 'Time'. The final hypothesis from the example would then be:

Hypothesis 6:
Begin context: <sp>
String: 'time' v 'Time'
End context: {x| x = space or x ∈ Punc.}

Once this hypothesis has been generated, it is then used to examine the input for negative examples that can strengthen or weaken the hypothesis. Suppose the input contained the fragment "... timely results..." . Processing the input with Hypothesis 6 would show a match for the string, but the end context would not agree; therefore, the hypothesis will be strengthened by changing the end context as shown below:

Final Hypothesis:
Begin context: <sp>
String: 'time' or 'Time'
End context: {x|x=space v
x ∈ Punc. &
x ∈ small letters}

After the input has been processed and a final hypothesis proposed, the hypothesis is used to construct a procedure such as shown in Figure 35. The first part of the procedure to be constructed is the transitions for the beginning context. The states in the procedure are the

instructions in the instruction set, and the arc labels consist of the information in the final hypothesis. A start state is placed in the procedure with an arc to a move right instruction (R). Since the procedure is a string match or look-ahead routine all states other than the start state will be move right instructions. Each of the states will have two arcs exiting them. The labels on these two arcs will be the negation of the each other.

The construction is accomplished by placing the first character of the begin context on the exiting arc going to a new move right state. The other arc is labeled with the negation of the character and this arc terminates at the first move right state. Each character of the begin context creates another move right state labeled as mentioned.

The string from the hypothesis is then used to complete the procedure that has been partially constructed. If the string is composed of disjunctions, the characters are used to form disjunctions. Each of the disjunctions are combined with conjunctions. The final hypothesis above provides a string of 'time' or 'Time'. The conjunction of disjunctions will be formed as:

('T' v 't') & ('i' v 'i') & ('m' v 'm') & ('e' v 'e')

Upon reduction the string will be expressed as:

('T' v 't') & 'i' & 'm' & 'e'

Each disjunction becomes a label on an arc to a new move

right state and the negation becomes the label on an arc back to the original move right state.

Finally, the end context is added in the same manner as the begin context. The first character becomes the label on the last move right state created from the string and new states are added for each character in the end context. The result of these operations is displayed in Figure 35.

IV. CONCLUSIONS AND RECOMMENDATIONS

A. SYNTHESIZER

The synthesizer that has been implemented for this thesis will produce programs from example computations in a reasonable amount of time. The system response for most of the traces was within 10 seconds on a Digital Equipment Corporation PDP-11/50 minicomputer. The response time is a function of the length of the trace and the number of multiple occurrences of a particular instruction or set of instructions in the final algorithm, with multiple occurrences of an instruction affecting response time the most. As Biermann [17] has noted, this has a nice implication for programming by example because most algorithms do not exhibit the characteristic of having a large number of instances of the same instruction. In other words, almost all multiple occurrences of an instruction in an input trace are indicative of a loop in the algorithm.

In all of the test cases except those that required a large amount of backups, static processing accounted for at least half of the total response time. Future modifications to the synthesizer which would decrease the total response time could be directed toward designing the static processing stage more efficiently. However, the trade-off between static processing and dynamic processing must be

kept in perspective. Static processing is a linear function of the length of the trace, whereas dynamic processing, since it is an enumerative search technique, is an exponential function of the length of the trace.

Another area which should be considered is the dynamic processing stage. There exists a plethora of research questions within this area. The primary one being: Can more information be gleaned from the input trace during static processing which will decrease the search time for dynamic processing? Difference sets and couple-classes provide some powerful mechanisms for decreasing the amount of search; however, lower bounds computations on the number of states required by the machine often increase the amount of search. Lower bounds are restrictive in nature. They are designed to force the final algorithm into a minimum state configuration which, in many cases, causes extra search time. Relaxation of the lower bounds computation will result in a final algorithm which may not be expressed in a minimum number of states, but which will still be deterministic. There might be better methods of initially computing the number of states which would result in a closer estimate of the actual number of states required for the algorithm. Obviously, the closer the initial guess is to the actual requirement, the less backup incurred, and, therefore, the less search time required.

Since the amount of search required is governed by the failure memory entries, the more dense the failure memory can be made, the more directed the search becomes. So another area for research is to determine if more information exists in the failure memory entries than is currently being used. How much information do the structure factor and the free state factor provide? Is there another factor which would be useful?

Finally, a more general question can be addressed. The underlying structure of this technique is an enumerative search. Can the technique be generalized to include other algorithms which are enumerative in nature? What modifications to the failure memory are needed? How would difference sets and couple-classes be redefined?

B. CONDITION PROCESSING

The condition processor front-end to the synthesizer relieves the user from worrying about some of the control structure considerations by automatically generating conditions. Another addition which would increase the power of the synthesizer is an automatic loop variable generator as discussed by Biermann [18]. Although the text editing environment has been used in this thesis work, the part of the condition processor design which deals with a context free environment is general enough that it could be designed to operate in any domain.

Condition generation in a context sensitive environment is a much harder problem further complicated by requisite pattern matching and pattern generation. Before this type of condition generation can be generalized, much work has to be done to increase the efficiency of pattern generation schemes. Angluin [19] has shown a pattern generation scheme which is a polynomial time algorithm for pattern generation with one variable, but the domain we have examined will require at least two variables. There is not a polynomial time algorithm for pattern generation with two variables. Heuristic techniques will probably be necessary to provide methods of pattern generation which will be fast enough to be useful over a wide range of problems.

APPENDIX A

```
/*  The command line needed to run the synthesizer is:

    a.out num < fn

        where 'a.out' is the executable file created by the
                    C compiler (%cc synthesizer.c )

              'num' is an integer digit between 0 and 9, inclusive
                      This provides a variety of output which
                      allows the user to follow the synthesis.

              '<' pipes the input file 'fn' to the program

              'fn' is the file name for the input trace

*/
/* For the current configuration of the synthesizer
    the input file must be in the following form:

              S  c0 I1
              I1 c1 I2
              I2 c2 I3
              I3 c3 I4
                  .
                  .
                  .
              In+1 c In
              In
```

```
        where I1 through In are the instructions and
              c1 through cn are the conditions
              S is a unique start symbol
              c0 is a null condition (e.g. '0' or space)

        The instructions and conditions can be encoded using
        any printable ASCII character.
        The coding must be consistent throughout the trace.

        Consistent representation means that all duplicate instructions
        in the input trace are coded with the same ASCII character, and
        all duplicate conditions are coded with the same ASCII
        character.

        NOTE that the file must end with a single
                 instruction.  The final instruction
                 in the input trace.
    */

    /*  These are the constant values which are used throughout the
        program.

        MAXCNT is the maximum length allowed for the input trace
        MAXVCTR is the maximum number of difference set elements which
                can be handled by the synthesizer
        MAXINST is the maximum number of different instructions allowed
                in the input trace
        MAXMEM is the maximum elements of a single couple-class

        FORCED, ARBITY, NEW, and OLD are possible state descriptors
    */

    #define EOF '\0'
    #define EOLN '\n'
    #define MAXCNT 200
    #define MAXVCTR 20
```

```
#define MAXINST 60
#define MAXMEM 40
#define TRIPLE 3
#define FORCED 'f'
#define ARBITY 'a'
#define NEW 'n'
#define OLD 'o'
#define NULL 0

/*  These are the global variables used throughout the program */

int triplecnt;                  /* num of triples from input file */
int Maxlabel;                   /* used num of col reqd in FM */
int wklvl;                      /* current working level */
int dbug;                       /* variable level debug stmts */
int contradiction;              /* boolean used to escape recursion */
struct
{
    char N;                     /* condition */
    char O;                     /* instruction */
    int Class;                  /* couple class */
    int DiffSet[MAXVCTR];       /* difference set */
    int Selector;               /* label of state */
    char Mode;                  /* label forced or arbitrary */
    char State;                 /* new state indicator */
}
TraceTable[MAXCNT];
int indx[MAXCNT];               /* xref from instr set to TraceTable */
struct
{
    int W;                      /* structure factor */
    int G;                      /* free state factor */
}
FM[MAXCNT][MAXVCTR];
int FMcntr[MAXCNT];             /* count of num col used in FM */
int FrStLim;
```

```
int FrStCnt;
struct
{
    char iname;              /* instruction */
    int L;                   /* lower bound on num of states */
}
Instruction[MAXINST];
int AddL[MAXINST];           /* used to incr instr state cnt */
main(argc,argv)

int argc;
char **argv;

/*  The program consists of two major parts
        Static processing is done on the trace to set up the
        Trace Table with couple class information, difference set
        information, and lower bounds on the number of states
        required in the final machine.

        Dynamic processing is the actual search algorithm used to
        assign state labels to the trace in such a way that the
        final product is a deterministic machine with the minimum
        number of states.
*/

{
    int i,j;             /* index variables */

    dbug = atoi(argv[1]);

    STATIC();

    DYNAMIC();

    if(dbug <= 9)
    {
```

```
            printf("TRACE TABLE \n\n");
            printf("LEVEL  TRANSITION   STATE \n\n");
            for(i=1;i<triplecnt;i++)
                printf("%2d          %c          %2d%c \n",
                i,TraceTable[i].N,TraceTable[i].Selector,TraceTable[i].O);
        }
    }
```

```
STATIC()

{
    int i,j,k,m,n,p,q,l;          /* general purpose indices */
    int save,compare;             /*  used for lower bound determine */
    char c;                       /* temp symbol holder */
    char temp[TRIPLE+1];          /* temp store for triples */

    struct
        {
        char symbol[TRIPLE+1];    /* input triple */
        int elements;             /* num of elmnts in couple class */
        int class;                /* couple class of triple */
        int member[MAXVCTR];      /* members of couple class */
        int diffset[MAXVCTR];     /* difference set members */
    }
    triples[MAXCNT];

    int memptr[MAXCNT];           /* pointer to next empty diffset elmnt */

    int check;                    /* index used during dynamic proc */

    int K;                        /* num of diff instr */
    int Statelimit;               /* max num of states permitted */
    int tempstore[MAXVCTR];       /* aids in count of L */


    /* Initialize all tables and variables other than indices */
    for(i=0;i < MAXCNT;i++)
    {
        memptr[i] = 0;
        indx[i] = 0;
        Instruction[i].L = 0;
        AddL[i] = 0;
        triples[i].elements = 0;
        triples[i].class = 0;
```

```
        TraceTable[i].N = ' ';
        TraceTable[i].O = ' ';
        TraceTable[i].Class = 0;
        TraceTable[i].Selector = 0;
        TraceTable[i].Mode = ' ';
        TraceTable[i].State = NEW;
        for(j=0;j < MAXVCTR;j++)
        {
            triples[i].diffset[j] = 0;
            triples[i].member[j] = 0;
            TraceTable[i].DiffSet[j] = 0;
        }
        for(j=0;j < TRIPLE+1;j++)
            triples[i].symbol[j] = ' ';
    }
    /* Read the triples from the input file and place them in
        triples.symbol for static processing
    */
    i=1;
    do
        {
        k=1;
        while(((c=getchar()) !=  EOLN) && (c != EOF))
            triples[i].symbol[k++] = c;
        i++;
    }
    while(c != EOF);
    triplecnt = i-1;

    /* Do the static processing
        Establish Difference Set for each level of the trace
        Establish Couple Class for each level of the trace
        Determine a lower bound on the total number of states required
            for the trace
        Determine a lower bound on the number of states for each
            instruction
```

```
    Determine the number of different instructions
*/

j = 1;
m = 1;
while(j < triplecnt)
{
    for(k=1;k <= TRIPLE;k++)
        temp[k] = triples[j].symbol[k];

    /* Compare each triple with those triples below it
        If all three elements match put the two triples into the
            same couple class.
        If the first two elements match, but the last element doesn't
            put the index of the triple[i] into the difference set
            for triple[j].
    */

    i =j+1;
    n = 0;
    p = 1;
    while(i < triplecnt)
    {
        k = 1;
        while(temp[k] == triples[i].symbol[k] && k <= TRIPLE)
        {
            ++k;
            if(k > TRIPLE)
                if(triples[j].class == 0)
                {
                    /*
                       triples.element keeps count of the number of
                            elements in the couple class
                       triples.member contains the level number of
                            each element within the same class
                        triples.class at this point in the program is
```

```
                                used as a check to insure all levels
                                have been compared
                        */

                        ++triples[j].elements;
                        triples[m].member[0] = j;
                        triples[m].member[p++] = i;
                        triples[i].class = 1;
                    }
            }


            /*   The first two elements match but the third element doesn't
                 memptr is a cross reference to keep track of the next
                 cell to fill in the difference set.
            */

            if(k == TRIPLE)
            {
                triples[j].diffset[n++] = i;
                memptr[j] = n;
            }
            i++;
        }
        if(triples[j].elements > 0)
            m++;
        j++;
    }
```

```
/*
Enter couple class index into triples table for all triples
A entry of 0 indicates a singleton triple
*/

n=0;
while(n <=m)
{
    p=0;
    while(triples[n].member[p] != 0)
    {
        triples[triples[n].member[p]].class = n;
        p++;
    }
    n++;
}

/*  Check for transitivity in the difference set */
/*  If level j is a member of DV for level i and levels i-1 and
    j-1 are in the same couple class, j-1 must also be a member
    of the DV for i-1
*/

for(m=0;m<MAXVCTR;m++)
    for(n=0;n<triplecnt;n++)
        if(triples[n].diffset[m] != 0)
        {
            i=n;
            k=triples[n].diffset[m];
            while(triples[i-1].class != 0 &&
                triples[i-1].class == triples[k-1].class)
            {
                p = 0;
                while((p <= memptr[i-1])
                    && (triples[i-1].diffset[p] != k-1))
                    p++;
```

```
                    if(triples[i-1].diffset[p] == NULL)
                        triples[--i].diffset[memptr[i]++] = --k;
                    else
                        {
                        i--;
                        k--;
                        }
                }
            }


    /* Given the input triples contained in triples.symbol, find
        the different instructions and keep a count of the number
        of different instructions.  Since the middle element of the
        triple is a condition and since the last element of the
        triple is a duplicate of the first element of the succeeding
        triple, it is sufficient to look at the first element only
    */

    /*  Initialize instruction set */

    for(j=0;j < MAXINST;j++)
        Instruction[j].iname = ' ';
    K=0;
    k = 1;
    for(i=1;i < triplecnt;i++)
    {
        c = triples[i].symbol[1];
        j = 1;
        while(Instruction[j++].iname != c && j < MAXINST);
        if(j >= MAXINST)
            Instruction[k++].iname = c;
    }
    K = k-1;

    /*
```

```
Static information gives the capability to compute lower bounds
on the number of states needed for each instruction, and by a
summation of those bounds, a lower bound on the total number of
states needed in the program in order to construct a deterministic
machine from the program trace.
*/

/*  Pick each instruction out of the instruction set in turn,
    and trace through the triples until a match is found.
    If the level that the match occurred doesnt have a difference
    vector, then the lower state bound on that instruction is 1.
    If a difference vector exists, the lower bound can be found by
    establishing a set of triples' levels which has no duplicates.
*/
Statelimit = 0;
j=1;
while(Instruction[j].iname != ' ')
{
    save = 1;
    l = 1;
    while(l < triplecnt)
    {
        compare = 1;
        if(triples[l].symbol[1] == Instruction[j].iname
            && triples[l].diffset[0] != NULL)
        {

        /* Put the difference set for level l into temp store */

            for(k=0;k<MAXVCTR;k++)
                tempstore[k]=triples[l].diffset[k];

            /*  As each element of the difference set is checked, if
                the element is in the same couple-class as any of the
                previous elements then the instructions are the same,
```

```
        so that element is discounted by entering
         -1 value in the tempstore location for that element
    */
    if(triples[i].class != NULL)
        for(k=0;tempstore[k] != 0;k++)
            if(triples[tempstore[k]].class ==
                triples[i].class)
                tempstore[k] = -1;

    n=0;
    while(tempstore[n] != 0)
    {
        for(k=n;tempstore[k]== -1;k++);
        for(m=k+1;tempstore[m]==-1;m++);
        while(tempstore[m] != 0)
        {
            if(triples[tempstore[k]].class != NULL)
                if(triples[tempstore[k]].class ==
                    triples[tempstore[m]].class)
                    tempstore[m] = -1;
            while(tempstore[++m] == -1);
        }
        while(tempstore[++n] == -1);
    }

    /*
    Once the set has been established, simply count the
    number of occurences in that set.  That is a lower bound
    on the states required for that instruction

    The number of occurences is counted by incrementing
    Instruction.L for each entry in tempstore that is > zero.
    */
    k=0;
    while(tempstore[k] != 0)
    {
```

```
                    if(tempstore[k] != -1)
                        compare++;
                    k++;
                }

                /* 'save' compares the value computed at each level for
                    a particular instruction and saves the maximum value
                    for the trace
                */

                save = (save > compare) ? save : compare;
            }
            l++;
        }
        /*  Statelimit is the lower bound on the total number of states
            needed for the machine
        */
        Instruction[j].L = save;
        Statelimit = Statelimit + Instruction[j].L;
        j++;
    }

    /* Determine the largest instruction state count.  Later this
        becomes the column size for the Failure Memory.
    */

    Maxlabel=Instruction[0].L;
    for(j=1;j<MAXINST;j++)
        if(Instruction[j].L > Maxlabel)
            Maxlabel=Instruction[j].L;


    /* Establish the TraceTable which will be used in the Dynamic
        processing of the instruction trace.  Since it is necessary
        to map the triples information from a three column structure
        to a two column structure, some offset must occur
```

```
    */
    TraceTable[1].N=' ';
    TraceTable[1].O=triples[1].symbol[1];
    for(i=0;i<MAXVCTR;i++)
        TraceTable[1].DiffSet[i]=triples[1].diffset[i];
    for(i=2;i<triplecnt;i++)
    {
        TraceTable[i].N=triples[i-1].symbol[2];
        TraceTable[i].O=triples[i].symbol[1];
        TraceTable[i].Class=triples[i-1].class;
        for(j=0;j<MAXVCTR;j++)
            TraceTable[i].DiffSet[j]=triples[i].diffset[j];
        TraceTable[i].Selector=0;
    }


/* Establish cross refer between Trace Table and Instruction set */
/*  The cross reference is established as indx[j].  Since C language
  does not have the capability to index an array by characters,
  indx[] accomplishes the same thing.  Each level of the trace has a
  value for indx which corresponds to the index for Instruction.iname
*/

    i=1;
    while(Instruction[i].iname != ' ')
    {
        j=1;
        while(j < triplecnt )
        {
            if(TraceTable[j].O == Instruction[i].iname)
                indx[j]=i;
            j++;
        }
        i++;
    }
}
```

```
DYNAMIC()

/*  The dynamic processing involves assigning values to the selector
variables.  This is done in conjunction with Failure Memory entries
providing the pruning of the search tree.

Three phases of dynamic processing are performed
        The Selector variable is assigned
        DifferenceVectors are resolved
        Dynamic equivalence is assured

at each point checks are made to insure that the search is still
producing a deterministic machine
*/
{

    int i,j;                    /* index variables  */

    /*  Initialize Failure memory */

    for(i=0;i<MAXCNT;i++)
    {
        FMcntr[i] = 0;
        for(j=0;j<MAXVCTR;j++)
        {
            FM[i][j].W = 0;
            FM[i][j].G = 0;
        }
    }

    /*  Initialize State Limit, State Count, and work level  */

    FrStCnt=0;
    FrStLim = 0;
    wklvl = 1;
```

```
    /*  Start search */

    while(wklvl < triplecnt)
    {
        contradiction = 0;
        Assign(wklvl);



        /*
The assignment is legal, so continue by resolving the
            difference set
        */

        if(TraceTable[wklvl].DiffSet[0] != NULL)
        {
            DVresn(wklvl,TraceTable[wklvl].Selector);
            if(contradiction)
                continue;
        }

        /*  Resolve dynamic nonequivalence if any exists  */

        if(TraceTable[wklvl].Class != NULL)
        {
            DynEquiv(wklvl,TraceTable[wklvl].Selector);
            if(contradiction)
                continue;
        }


        wklvl++;
    }
}
```

```
Assign(level)
int level;
{
    int n,j,k,l;            /* index variables */

    /*  Check for a forced assignment */

    /*  Assignment can be FORCED or ARBITRARY.  A forced assignment
        occurs when the instruction-condition-instruction sequence is
        in a couple class that has previously been seen, and the first
        instruction has the same label as the first instruction of the
        couple class.
        An arbitrary assignment occurs otherwise, and it is based on the
        first invalid Failure Memory,FM, entry at that level.
    */


    if(TraceTable[wklvl].Class != NULL)
    {
        j = wklvl-1;
        while(j > 0)
        {
            if(TraceTable[j].Class == TraceTable[wklvl].Class)
              if(TraceTable[j-1].Selector==TraceTable[wklvl-1].Selector)
                    /*  Forced assignment  */

                {
                    TraceTable[wklvl].Selector = TraceTable[j].Selector;
                    TraceTable[wklvl].Mode = FORCED;

                    /*  TraceTable.State is OLD or NEW.  That is, if the
                        state label-state name combination occurs earlier
                        in the trace, it is an old state.
                        This must always be the case with a forced
                        assignment.
                    */
```

```
                    TraceTable[wklvl].State = OLD;
                    j = 0;
                }
            j--;
        }
    }


    /*  If the assignment has resulted in a contradiction then
        backup and attempt a reassignment
    /*

        if(TraceTable[wklvl].Mode == FORCED)
            && (FM[wklvl][TraceTable[wklvl].Selector].W != 0
                && FrStCnt < FM[wklvl][TraceTable[wklvl].Selector].G))

            Backup();

    /*  Arbitrary assignment  */

    if(TraceTable[wklvl].Selector == NULL)
    {
        n=0;
        while(FM[wklvl][++n].W > 0);  /* find invalid FM entry */
        TraceTable[wklvl].Selector = n;
        TraceTable[wklvl].Mode = ARBITY;
        TraceTable[wklvl].State = NEW;
        for(l=0;l<wklvl;l++)
            if(TraceTable[l].O == TraceTable[wklvl].O)
                if(TraceTable[l].Selector == TraceTable[wklvl].Selector)
                {
                    TraceTable[wklvl].State = OLD;
                    l = wklvl;
                }
    }
```

```
        /*  If the assignment has caused a contradiction then
            backup and try a reassignment
        */

        if(TraceTable[wklvl].Selector > Instruction[indx[wklvl]].L)
            if(FrStCnt == 0)
            {
                Backup();
                Assign(wklvl);
            }
        else
            AddState(wklvl);
        return(0);
    }
```

```
Backup()

/*  Backup operates by reducting the current working level until a
    Fixup can be performed
*/
{
    int i;
    while((wklvl > 0)
        && ((TraceTable[wklvl].Mode == FORCED)
        || ((TraceTable[wklvl].Selector >= Instruction[indx[wklvl]].L)
        && (FrStCnt == 0))
        || (TraceTable[wklvl].State == NEW)
        || ((FMcntr[wklvl] >= Instruction[indx[wklvl]].L)
        && (FrStCnt == 0))))
    {
        TraceTable[wklvl].Selector = 0;
        TraceTable[wklvl].State = NEW;
        TraceTable[wklvl].Mode = ' ';
        wklvl--;
    }
    /*  If Backup causes the working level to decrement to zero, add
        anotner free state and restart tne assignments.
    */
    if(wklvl <= 0)
    {
        FrStLim++;
        FrStCnt = FrStLim;
        for(i=1;Instruction[i].L != 0;i++)
        {
            Instruction[i].L = Instruction[i].L - AddL[i];
            AddL[i] = 0;
        }
        wklvl = 1;
    }
    Fixup(wklvl,TraceTable[wklvl].Selector);
}
```

```
Fixup(level,label)
int level,label;

/*  Fixup attempts to correct the contradiction by incrementing
    the Selector to n+1
*/
{

    int i,j,l;                      /*  index variables  */

    for(i=0;i<triplecnt;i++)
        for(j=0;j<=Maxlabel;j++)
            if(FM[i][j].W >= wklvl)
            {
                FM[i][j].W=0;
                FM[i][j].G=0;
                if(wklvl <= 0)
                    FMcntr[i] = 0;
                else
                    FMcntr[i]--;
            }
    if(wklvl > 0)
    {
        FM[level][label].W=wklvl-1;
        FM[level][label].G=FrStCnt+1;
        FMcntr[level]++;
    }
    TraceTable[level].Selector = NULL;
    return(1);
}
```

```
DVresn(level,label)
int level,label;

/*  Difference Vector resolution prevents future assignments being
    made that are shown to be illegal from the static processing
*/
{

    int j,m,k,p,n,i;                    /* index variables  */

    j=0;
    while(TraceTable[level].DiffSet[j] != NULL)
    {
        k=label;
        m=TraceTable[level].DiffSet[j];

        /*  If the FM cell does not contain an entry from a previous
            level, make the entry;  otherwise, leave the previous entry
            in the cell.
        */

        if(FM[m][k].W <= 0)
        {
            FM[m][k].W = wklvl;
            FM[m][k].G = FrStLim+1;
            FMcntr[m]++;

            /* If the entry causes an incipient fence, the implication
               is that more states are needed;  therefore, go to the
               current working level and try a Fixup.
            */

            if(FMcntr[m] >= Instruction[indx[m]].L)
                if(FrStCnt == 0)
                {
                    Fixup(wklvl,TraceTable[wklvl].Selector);
```

```
            contradiction=1;
            return(1);
        }

    /* If free states are available, use one of them and continue
       with the resolution.
    */

    else
        {
        AddState(m);
    }

    /*  If entry causes only one space to be left in FM, pseudo
        assign results on the remaining space
    */
    if(FMcntr[m] == Instruction[indx[m]].L-1 )
        /*  Two actions are taken, fence prevention is
            accomplished for all levels between level m and wklvl
            which contain m in the DV,
            and psuedo assignment causes the same entries to be
            made in FM as a normal assignment would cause
        */

    {
        p=1;
        while(FM[m][p].W != NULL && p <= Instruction[indx[m]].L)
            p++;                        /* pseudo label */
        n=level+1;
        while(n < m)                /* fence prevention */
        {
            i=0;
            while(TraceTable[n].DiffSet[i] != NULL)
            {
                if(TraceTable[n].DiffSet[i] == m)
                    if(FM[n][p].W == 0)
```

```
                                {
                                    FM[n][p].W=wklvl;
                                    FM[n][p].G=1;
                                    FMcntr[n]++;
                                    if(FMcntr[n] >= Instruction[indx[n]].L)
                                      if(FrStCnt == 0)
                                      {
                                      Fixup(wklvl,TraceTable[wklvl].Selector);
                                      contradiction=1;
                                      return(1);
                                      }
                                    else
                                        {
                                        AddState(n);
                                    }
                                }
                            i++;
                        }
                        n++;
                    }
                    /* Difference set resolution on psuedo assignment  */
                    if(TraceTable[m].DiffSet[0] != NULL)
                    {
                        DVresn(m,p);
                        if(contradiction)
                            return(1);
                    }
                }
                DVDyn(m,k);
                if(contradiction)
                    return(1);
            }
            j++;
        }
        return(0);
    }
```

```
DynEquiv(level,label)

int level,label;
/*  Dynamic nonequivalence results when levels i and j are in
    the same couple class, but label j cannot equal label i because
    the FM entry at level j col(label i) is valid.  Therefore,
    label for level j-1 cannot equal label for level i-1
    */

{

    int i,j,k,p;                            /* index variables */


    /*  Dynamic Equivalence is checked by inspecting all levels below the
        current working level for equivalent couple classes.

        If the Failure Memory prevents the same labels being assigned to
        the elements of equal couple classes, then dynamic nonequivalence
        occurs, so an entry is made in the FM to prevent the assignment
        at a future time when the working level gets to that level of the
        trace.
    */

    k=triplecnt;
    while(TraceTable[--k].Class != TraceTable[level].Class
        && k > level);
    if(k <= level)
        return(0);
    for(j=level;j>0;j--)
        for(i=k;i>=level;i--)
            if(TraceTable[i].Class == TraceTable[j].Class
                && TraceTable[i].Class != 0)
                if(FM[i][TraceTable[j].Selector].W != 0)
                    if(FM[i-1][TraceTable[j-1].Selector].W == 0)
                    {
```

```
                        FM[i-1][TraceTable[j-1].Selector].W = wklvl;
                        FM[i-1][TraceTable[j-1].Selector].G =
                            FM[i][label].G;
                        FMcntr[i-1]++;
                        if(FMcntr[i-1] >= Instruction[inix[i-1]].L)
                            if(FrStCnt == 0)
                            {
                            Fixup(wklvl,TraceTable[wklvl].Selector);
                            contradiction=1;
                            return(1);
                        }
                        else
                            AddState(i-1);
                        if(FMcntr[i-1] == Instruction[indx[i-1]].L-1
                            && FrStCnt == 0)
                            {
                            p=1;
                            while(FM[i-1][p].W != 0
                                && p <= Instruction[indx[i-1]].L)
                                p++;
                            DynEquiv(i-1,p);
                            if(contradiction)
                                return(1);
                        }
                    }
    return(0);
}
```

```
DVDyn(row,col)
int row,col;
{
    int i;
    if(TraceTable[row].Class != NULL)
        for(i=row-1;i>0;i--)
        if(TraceTable[row].Class == TraceTable[i].Class)
            if(col == TraceTable[i].Selector)
            if(FM[row-1][TraceTable[i-1].Selector].W == 0)
        {
        FM[row-1][TraceTable[i-1].Selector].W = wklvl;
        FM[row-1][TraceTable[i-1].Selector].G = FrStLim+1;
        FMcntr[row-1]++;
        if(FMcntr[row-1] >= Instruction[indx[row-1]].L)
            if(FrStCnt == 0)
            {
            Fixup(wklvl,TraceTable[wklvl].Selector);
            contradiction=1;
            return(1);
        }
        else
            AddState(row);
        DVDyn(row-1,TraceTable[i-1].Selector);
        if(contradiction)
            return(1);
    }
    return(0);
}
```

```
AddState(level)
int level;
{
    /*  AddState increases the bound on the number of states allowed
            for a particular instruction when there are free states
            available to the machine and the instruction at the current
            level needs an extra state in order for the machine to remain
            deterministic
        */

    FrStCnt--;
    AddL[indx[level]]++;

    Instruction[indx[level]].L =
        Instruction[indx[level]].L + AddL[indx[level]];
    if(Instruction[indx[level]].L > Maxlabel)
        Maxlabel=Instruction[indx[level]].L;
}
```

```
/*  ATOI is used to provide a debug level for the program  */

atoi(s)
char s[];
{
    int i,n;
    n=0;
    for(i=0;s[i] >= '0' && s[i] <= '9';i++)
        n= 10*n+s[i]-'0';
    return(n);
}
```

LIST OF REFERENCES

1. Iverson, K., "Operators", ACM Transactions on Programming Languages and Systems, Vol. 1, No. 2, October 1979.

2. Dijkstra, E. W., A Discipline of Programming, Prentice Hall Inc., 1976.

3. Green, C., "The Design of the PSI Program Synthesis System", Proceedings Second International Conference on Software Engineering P. 4-18, October 1976.

4. Glass, R. E., Computing Projects Which Failed, Computing World, 1977.

5. Heidorn, G. E., "Automatic Programming Through Natural Language Dialog: A Survey", IBM J Res Dev 20, 336, July 1976

6. Biermann, A. W., Natural Language Processing, paper in preparation, Naval Postgraduate School, 1981.

7. Walker, D. E., Understanding Spoken Language, Elsevier North-Holland Inc., 1978.

8. Smith, D. R., "A Design for an Automatic Programming System", Proceedings of the 7th International Joint Conference on Artificial Intelligence, Vancouver, B. C., Canada, 1981.

9. Manna, Z. and Waldinger, R., "A Deductive Approach to Program Synthesis," ACM Transactions on Programming Languages and Systems, Vol. 2, No. 1, P. 90-120, January 1980.

10. Manna, Z. and Waldinger, R., "Synthesis: Dreams ====> Programs," IEEE Transactions on Software Engineering, Vol. SE-5, No. 4, July 1979.

11. Biermann, A. W., "Approaches to Automatic Programming", Advances in Computer Science, Vol. 15, P. 1-63, 1976.

12. Smith, D. R., "A Survey of the Synthesis of LISP Programs from Examples", Proceedings of the International Workshop on Program Construction, Bonas, France, 1980.

13. Summers, P. D., "A Methodology for LISP Program Construction from Examples", JACM 24, P. 161-175, 1977.

14. Biermann, A. W., "The Inference of Regular LISP Programs from Examples", IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-8(8), P. 585-600, August 1978.

15. Gold, E. M., "Language Identification in the Limit", Information and Control, Vol. 10, P. 447-474, 1967.

16. Biermann, A.W. and Krishnaswamy, R., "Constructing Programs from Example Computations," IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, P. 143-153, September 1976.

17. Biermann, A.W., Baum, R. I. and Petry, F. E., "Speeding up the Synthesis of Programs from Traces," IEEE Transactions on Computers, Vol. C-24, No. 2, P. 122-136, P. 122-136, February 1975.

18. Biermann, A. W., "Automatic Insertion of Indexing Instructions in Program Synthesis", International Journal of Computer and Information Sciences, Vol. 7, No. 1, March 1978.

19. Angluin, D., "Finding Patterns Common to a Set of Strings", Computer Systems Journal, Vol. 21, # 1, August 1980.

BIBLIOGRAPHY

Bibel, W., "Syntax-Directed, Semantics-Supported Program Synthesis", Artificial Intelligence, Vol. 14, P. 243-261, 1980.

Follet, R., "Synthesising Recursive Functions with Side Effects", Artificial Intelligence, Vol. 13, P. 175-200, 1980.

Hewitt, C.E. and Smith, B., "Toward a Programming Apprentice," IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, P. 26-45, March 1975.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52B7 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4.	Professor Douglas R. Smith, Code 52SC Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5.	Captain C. W. Miller, USMC 109 Arterburn Rd. Louisville, Kentucky 40222	1
6.	Captain J. S. Lape, USMC 6207 Doncaster Court Springfield, Virginia 22150	1

Thesis
M58574 Miller 193232
c.1 Condition recognition for a program synthesizer.